Lotus. software

IBM

Draft Document for Review February 7, 2002 6:36 am

SG24-6667-00

Working with the Sametime Community Server Toolkit

Enhance and extend Sametime services

Create your own Sametime service

Support new clients and more

Søren Peter Nielsen
Assaf Azaria
Yaron Reinharts
Firas Yasin

Redbooks

ibm.com/redbooks

International Technical Support Organization

Working with the Sametime Community Server Toolkit

February 2002

SG24-6667-00

Take Note! Before using this information and the product it supports, be sure to read the general information in "Special notices" on page vii.

First Edition (February 2002)

This edition applies to the Sametime Community Server Tookit that is delivered with Sametime 2.6. The book is written based on a pre-release of this toolkit.

This document created or updated on February 7, 2002.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. TQH Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Contents

Special notices

IBM trademarks

Other company trademarks

Preface

The ability to create Sametime server side applications dramatically increase the potential for great benefits from using real time collaborations in all parts of businesses and organizations.

This redpiece contains some of the content that will be included in the upcoming Redbook about the Sametime Community Server Toolkit. It is being released in a very early stage to provide additional information for the users of the beta version of the Sametime Community Server Toolkit.

We introduce the Sametime server architecture which you need to understand to get the most out of server side programming.

We discuss two of the samples included with the toolkit called HackersCatcher and PlacesLogger.

Finally we take you through how to create a new Sametime service called SportsUpdater. This is also included as a sample in the beta version of the toolkit.

The beta version of the Sametime Community Server Toolkit is available for download from

`http://www.lotus.com/sametimedevelopers`

Note: This is a Redbook-in-progress that we release in draft state to allow our readers to get the information as soon as possible. It has not been through editorial check and you may experience grammatical error, layout error and so on. These things will be checked in the final version of the Redbook. However, we welcome feedback on the technical content of this redpiece.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Cambridge Center.

Søren Peter Nielsen works for the International Technical Support Organization at Lotus Software - IBM Software Group, Cambridge, Massachusetts. He manages projects that produce redbooks about all areas of Lotus products. Before joining the ITSO in 1998, Søren worked as an IT Architect for IBM Global Services in Denmark, designing solutions for a wide range of industries. Søren is a Certified Lotus Professional at the Principal level in Application Development and System Administration.

Assaf Azaria is an architect of the Sametime Community Server Toolkit in IBM Rehovot, (Ubique) Israel. Assaf has been working as a member of the Sametime team for more than four years as one of the architects of the Sametime server and client toolkits.

Yaron Reinharts is the Sametime Community Server team leader, in IBM Rehovot, (Ubique) Israel, he has been working as a member of the Sametime team for more than three years, as one of the architects of the Sametime Server and the Sametime toolkits, Yaron and his team created the Sametime Community Server from scratch.

Firas Yasin is a graduate in Computer Science from the University of Kentucky and holds a masters in Public Administration from the Martin School of Public Policy. He is a Software Engineer at the IBM software group in DataBeam. His role is to provide custom applications and customer support for Team Collaboration Technologies including Sametime and QuickPlace.

We want to extend a special thank you to Harry Hornreich that have contributed greatly to the book with managerial and other support:

Harry Hornreich is development manager for the Sametime Toolkits and Clients in the Haifa Software Lab, Israel, since August 1997. In this role he has driven the development of the Java, C++, Sametime Links,COM and Community Server Toolkits. In addition he is responsible for the Sametime Connect clients which are built on top of these Toolkits. Harry has a M.Sc. in Computer Science from the Technion in Haifa and has more than 12 years experience in software development.

We also extend a special thank you to **Jeremy Dies** - Lotus Sametime brand manager and **Don Bunch** - Lotus Sametime product manager for their help and support.

In addition we send thanks to the following people for their contributions to this project:

- Christopher Baker, IBM Westford Software Lab
- Dvir Landerer, IBM Haifa Software Lab, Israel
- Amir Perlman, IBM Haifa Software Lab, Israel

- Haim Schneider, IBM Haifa Software Lab, Israel
- Marjorie Schejter, IBM Haifa Software Lab, Israel
- Yafit Sami, IBM Haifa Software Lab, Israel

Notice

This publication is intended to help architects and developers to design and develop Sametime server side solutions. The information in this publication is not intended as the specification of any programming interfaces that are provided by the Sametime Community Server Toolkit. See the PUBLICATIONS section of the IBM Programming Announcement for Sametime 2.6 for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to the address on page ii.

1

Introduction

This chapter is not included in this redpiece.

2

Sametime server architecture

This chapter describes the basics of the Sametime server architecture.

In this chapter, the following topics are discussed/described:

- The user model of the Sametime community
- The Sametime server structure
- Communication in Sametime
- Distribution, scalability and redundancy
- Multi Server solutions
- Connecting between communities

2.1 The user model of the Sametime community

The Sametime user and login model supports multiple users running multiple applications concurrently.

The user model is divided into persistent and runtime parts.

2.1.1 Persistent user data

Each user in the ST community has the following basic properties:

- User ID – A string representing persistent & unique identifier of a user in the community.
- Login parameters – Set of fields used for creation of a login into the community. Common fields will be a login name and a password. Where the login name is a string that has no further usage beyond the login phase.
- User Name – A name under which other users know the user. A user might have different user names for different logins.
- Description – A descriptive text about the user. Again, a description may be different for different logins.
- Privacy List – A list of user IDs that may or may not know that the user is online. A privacy list is maintained per user. If a user logs into the community several times simultaneously, the server synchronizes the privacy list of the user between its different logins.

The above model allows for a user to have multiple names and to still be known as the same user by other community members regardless of the login name s/he uses. This mechanism is called aliasing.

2.1.2 Runtime user structures

The runtime user model of ST enables each user to create multiple concurrent logins to the community.

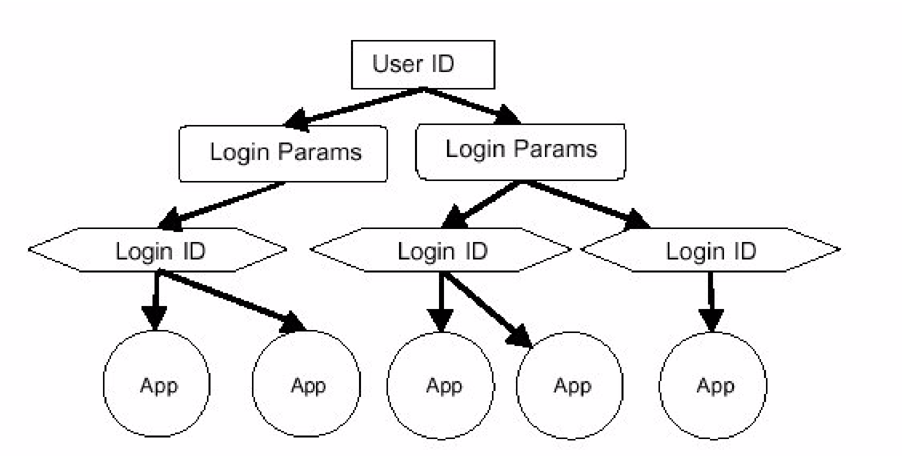


Figure 2-1 Run Time User Model

Each TCP/IP connection of a user is regarded as a single login of the user to the community. A login ID that is unique in all the community is assigned to each connection.

Even though all the applications of a user can use the same login, multiple logins are needed when running non-cooperative (e.g. Java and OCX components) applications. Currently Sametime communities limit the user logins to a single machine (single IP address). This is done in order to ensure good user experience.

2.1.3 Guests

A guest is a non-authenticated user of the community. A community administrator may configure what services of the community are available for guests. A guest does not have a persistent user ID and only the login ID can be used for addressing the guest. Furthermore there is no way to associate between different logins of the same guest person.

2.1.4 Inter community identifiers

In order to enable interactions between communities a global unique identifier of users and logins (or any other entity in the community) is needed. A unique community ID is defined for each community. Attaching the community ID to the user & login IDs ensures their uniqueness between communities.

2.2 The Sametime Server Structure

A Sametime community is composed from the following layers. Layers are listed bottom up:

- Clients – A program that logs into the community as a community user.
- Multiplexers – Mediates between clients and a server
- Server – The core server of the community.
- Server Applications – Supply add-on service to the community.

A community is a collection of such server, either in a distributed (WAN) and/or scalable (LAN) environments.

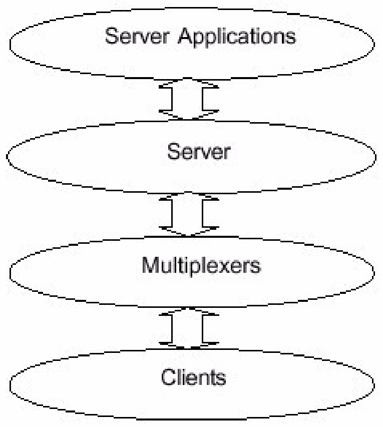


Figure 2-2 Sametime Community Layers

2.2.1 Multiplexers Layer

Multiplexers are I/O concentrators that enable powerful Sametime servers by the following means:

- I/O concentration – Concentrates I/O from clients toward the Sametime server. This concentrationlimits the I/O overhead of the server since it has to send and receive data on a small number of connections. Hence, the overhead of connection initialization, polling and termination is much reduced.
- Distribution of multi recipient messages – The server is able to send a single copy of a message (with a recipient list attached) to its multiplexer. The multiplexer then distributes the message to the indicated recipients.
- Gateway – The Multiplexer can act as a gateway; translating protocols between clients or third party server software and the Sametime server. An example of such a multiplexer is an extended HTTP multiplexer that allows clients to work with the Sametime community using the HTTP protocol thus solving firewall problems. Acting as a gateway, the multiplexer can allow integration of different protocols while enabling the Sametime server to continue and use the Sametime optimized protocol.

The multiplexer layer is transparent to the clients that connect the community through them. Multiplexers can connect in a multi-level where a multiplexer is not directly connected to the server but connects to another multiplexer in a chain of multiplexers. Layering can be used for localizing users connections and./or in a firewalled environment where the multiplexer acts as a proxy.

The Sametime Server Toolkit enable you to write your own Multiplexers.

2.2.2 The core server

The server who is the core of the Sametime community is responsible for the following:

- Manages the community members (users, logins, multiplexers & server applications).
- Routes messages
- Supplies notifications on its community members

2.2.3 Server application layer

Server applications enhance the functionality supplied by the server. A server application connects to a server and declares the services it is supplying. The server routes requests for such services to the appropriate instance of the server applications.

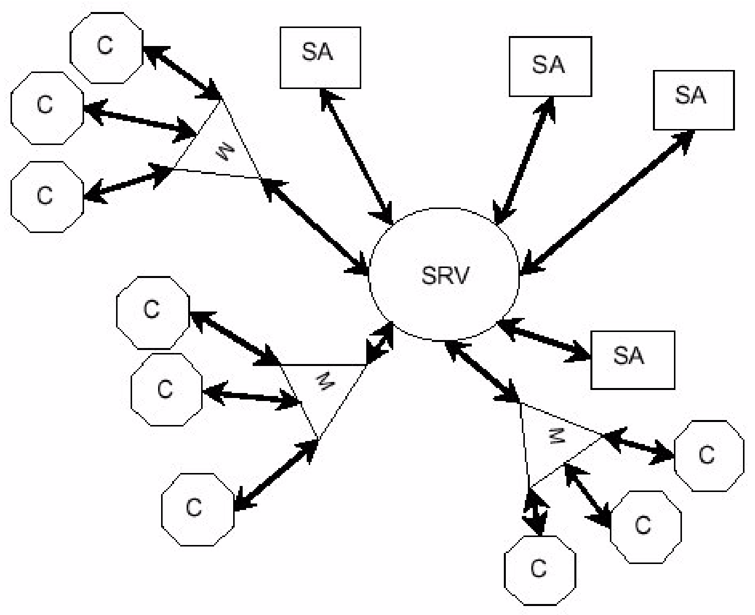
Following services are supplied by native Sametime server applications. Third party vendors using the Server toolkit will be able to develop other services:

- Who Is Online (WIO) – Widely known as the buddy list service. This service supplies users with notifications about status & properties of other users in the community.
- Authenticate – Works against the database and authenticates users for its server.
- Resolve – Resolves a given full user name to a list of matching user IDs.
- Browse & Search – Browses and/or searches the user directory give some user properties, e.g. user address.
- N-way chat – Supplies chat-rooms where several users can participate.

Note that Instant Messages (IM) are supplied directly by the Sametime server.

The Sametime Server Toolkit lets you write your own Server Applications and add your server side logic to the community.

2.2.4 Sametime community schematic view



SRV – Server, M – Multiplexer, SA – Server Application and C – Client.

Figure 2-3 Sametime community schematic view

The server is launched and multiplexers and server applications are connected to it via TCP/IP connection. Clients are connected to one of the multiplexers also via TCP/IP connection. The server assigns login ID and handles the properties of each community member connected to it. Authentication of users is done via server application supplying the authentication service. This server application works with the database while the server does not access the database directly.

2.3 Communication in Sametime

In this section we'll describes the communication mechanisms in a Sametime community.

2.3.1 Addressing modes

There are several ways to address other community member:

- User ID – Some user is addressed. If the recipient user has more than a single login, one of its logins is selected.
- Login ID – A particular login is addressed. This time no selection of logins is needed.
- Service Type – A server application providing the specified service type is addressed. There might be multiple instances of providers of this service. The server has to decide (while possibly consulting the available providers) which provider will server the request.

2.3.2 Addressing scopes

The most frequent type of interaction in a community is one to one. Messages are interchanged between two community members. In addition to the above there are multi recipients types of interactions that are candidates for performance optimizations. Sametime communication supplies the following methods for such interactions:

- One to Many – The message is intended to multiple recipients. Common case is N-way chat where a sentence sent by one member in the chat has to reach all chat participants. Only a single instance of the message has to be sent from the source. The message contains a list of recipients. The server distributes a single copy of the message to the multiplexers involved while the multiplexers distribute the message to the logins involved.
- Broadcasts – The message is broadcast to all community members. The server sends a single copy of the broadcast message to each multiplexer connected to it; each multiplexer dispatches the message to its logins.

2.3.3 Channels

Routing of messages is typically done via virtual connections called a channel. A channel may span several TCP/IP connections. For example when a client on multiplexer A creates a channel to a client on multiplexer B, the channel traverses the following TCP/IP connections:

- The connection from the user login to multiplexer A
- The connection from multiplexer A to the server
- The connection from the server to multiplexer B
- The connection from multiplexer B to the login of the other user.

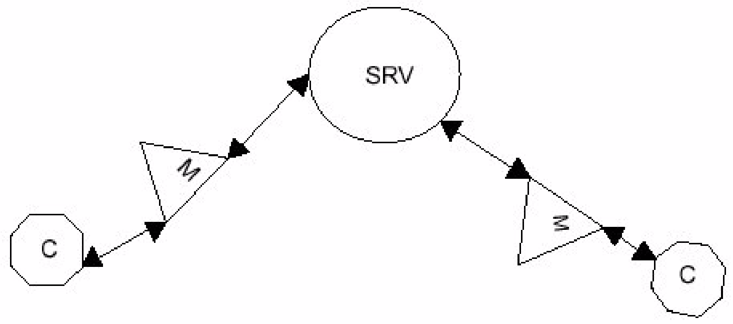


Figure 2-4 An open channel

As described above, the network route between any community members may involve several hops. The channel ensures the order of messages in an interaction and ensures connectivity by providing a notification when the route of the channel is broken (e.g., when a crash occurs on one of the parties or on one of the community members along the channel route). Both channel participants receive a notification when the channel is broken. Data that enables routing of channel messages is stored in each of the community entities of the channel route.

The Sametime toolkit supply the Channel class which encapsulates all the abilities of a Sametime channel.

2.4 Distribution, scalability and redundancy

Until now we have described a Sametime community consisting of a single core server. This solution is adequate for many communities. A single Sametime server is designed to serve about 100000 simultaneous logins. However a singles server can not address some situations that are described below.

2.4.1 Distribution

Community users may be distributed over remote geographical sites. In a single server environment remote clients depend on the network quality between their site and the server. This means that if the network path to the server is temporarily down, remote users can not get services to interact even with nearby users. In addition, network utilization is not optimized; a one to many or broadcast message is sent for each client separately. More common example is the case of an IM between two neighboring users connected via remote server; the message has to travel to the remote server and back. A local multiplexer connected to the remote server addresses the issue of multi recipient messages however it does not address the connectivity issue.

2.4.2 Scalability

A single Sametime server has its upper limits. While these limits are valid for most organizations they do not match expectations in very big organizations (e.g. IBM) and in the services market (e.g. Excite) Sametime has to provide a solution for serving millions of users under the same community.

2.4.3 Redundancy

In orthogonal to the issues of distribution and scalability, the dependency on a single server machine is still a problem. Users should be able to connect to the community even when some server(s) crash or they are in maintenance.

2.5 Multi server solutions

Following is a schematic view of a multi server Sametime community. Note that the only schematic change is having several servers connected to each other as the community core instead of single server. Users are not affected by connecting to a single or multi server community. However, various technological problems arise in a multi server community as routing of messages, addressing of community members and more. These problems are addresses in this section.

2.5.1 Routing

A channel or a message destined to other community member may need to traverse several servers and a route has to be chosen. In current versions of Sametime we avoid this problem by connecting every server in the community to every other server. This type of connection is known as a clique. The route between any community members will traverse at most two servers: the server into which the first community member is connected and the server into which the second community member is connected.

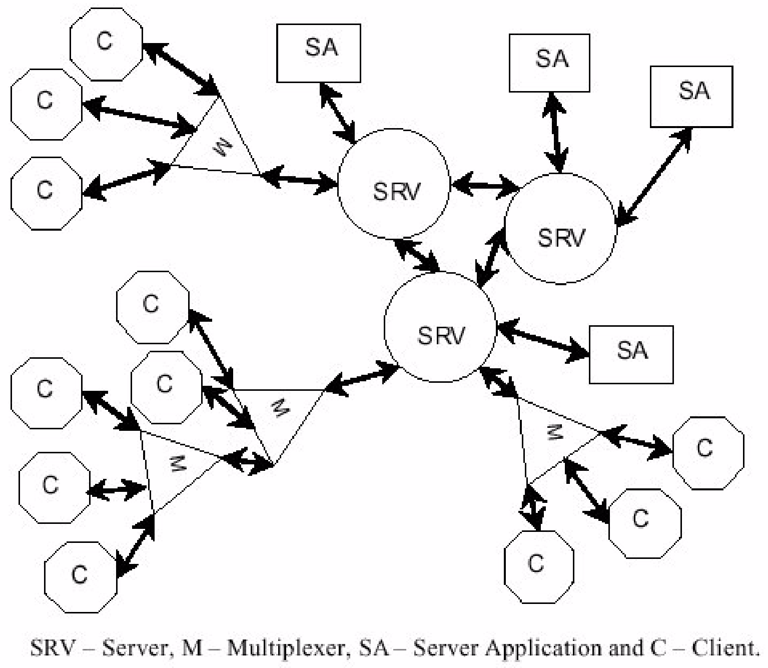

C
C
C
M
SA
SRV
SA
SA
SRV
SRV
SA
C
C
C
C
C
C
M
M
M
C
C
C
SRV – Server, M – Multiplexer, SA – Server Application and C – Client.

Figure 2-5 Routing message between servers

2.5.2 Addressing

In a single server community when a member is addressed either by a user ID, by a login ID or by a service type, the server routes the message to the appropriate community member as it appears in its local tables. In a multi server environment it is not possible. Various solutions are possible for this addressing this problem.

The solution implemented in Sametime is as follows. A global knowledge of all the community is maintained by dedicated server applications. This server application is the online directory (OLD). Each server has an instance of the OLD. The online directory gathers and maintains minimumdata that enables locating online entities (e.g. users) in the entire community. It is important to understand that minimum data gives enough information only for locating an entity and subsequent queries to other community members are needed for gathering all the information needed. The server notifies its OLD about all otherservers in the community. The OLD creates a channel to every other OLD in the community and gets the current state and subsequent updates from them. Eventually all the online directories replicate the data of each other.

Given the online directory, when a server has to locate a community member by user ID, it simply consults its OLD.

When a community member is addressed by a login ID, the services of the OLD are not needed. A login ID is contains the IP address of the server in which the login resides. The server simply extracts the IP address and contacts the server indicated by it.

The services of the OLD are very much needed however, when addressing a community member by service type. Part of the information maintained by the OLD are the various service providers in the community and the server to which each one is connected. When addressing by a service type, the server consults the online directory for selecting between the possible service providers.

2.5.3 Persistent Storage

User properties are kept in a persistent storage that is associated with each server; this storage is replicated between servers. Since in a geographically distributed environment it is not possible to replicate this data in real-time, a user that has changed its properties in one server and then logs into a different server might get an out of date data

In order to solve the above problem a home server is defined for each user in the community. This is done by attaching an address of a server to each registered user in the community. This server will be the home server for the user it is attached to. The client program may get the home server from its local storage or try and connect to some server in the community. If the connection is established to a different server than the defined home server for the user, the connection is internally redirected, using the channel mechanism, to the defined home server of the user.

2.5.4 Scalability & Redundancy

A multi server community as described above can solve the distribution issue. The scalability & redundancy issues are not solved yet. For addressing these issues the notion of cluster is introduced. A cluster is a set of servers where every server in the cluster is capable of behaving as the home server of all users of the cluster. That is a home cluster is defined for the users of the cluster.

2.6 Connecting between Communities

Until now we have described a single Sametime community. Many times community users would like to get services from different communities. E.g. user in community Mars would like to be aware when his friend/colleague from community Venus is online.

Sametime enables users in one community to get services from other Sametime communities

Following are some details regarding community to community connections.

One server or more in each community are designated as the “foreign affairs ministers”. These servers are able to connect to other communities on behalf of their community. The administrators of each community determine to which communities their community can connect and what services are available for other community users.

Each Sametime community has a community ID that is global between all Sametime communities. This ID is used for enhancing the user and login ID, which are unique in a single community to be unique over all Sametime communities.

User privacy between communities is maintained. Each community is notified and enforces privacy restrictions defined by users of other communities. For example user Joe in community Mars can define that user Bill from community Venus may not be aware on him. This privacy is maintained when communities Mars and Venus are inter-connected.

3

Places architecture

In this chapter we will discuss the Sametime places architecture with emphasis on the server side capabilities. The reader is recommended to read the Places Architecture chapter of the Sametime Java toolkit redbook before reading this document. It will be assumed that the concepts that were discussed there are understood. We will also not delve into the technical aspects of how to use the toolkit. The developer that is interested in technical information is encouraged to take a look at the documentation that is provided with the Sametime server toolkit.

In this section we will discuss

- The places server application
- Configuration of places
- The places directory
- Activities
- Permissions

3.1 The places server application

Sametime provides its services using autonomic server side applications. The places services make no exception. The places server application provide different capabilities to server application and to clients. Those different capabilities are exposed in the server toolkit in the PlacesAdminService and the ActivityService components.

3.2 Configuration of places

When a client application creates a virtual place, its characteristics and behavior is somewhat predefined. For example it always has exactly three sections and it is always a non – persistent place. A server application that uses the PlacesAdmin component of the server toolkit is able to create persistent places (places that exist regardless of the existence of users inside them) and to configure the number of sections to be created for a place of a given place type. (If you do not understand what a place or section are you should read the Place Architecture chapter of the client Java toolkit redbook).

For a sample of using persistent places and the admin component refer to the AuctionHouse sample (NOT INCLUDED in this redpiece).

3.3 The places directory

Another capability that is provided by the places SA to server applications is a sort of a directory for existing places. Registering to receive directory notification will provide information on newly created places, on places that were destroyed and on changes that happens on an exiting place. First usage that comes to mind is a chat room that is divided to places according to different discussions that go on there. Using the places directory a navigation pane can be created allowing users to select the ‘room’ they want to go into.

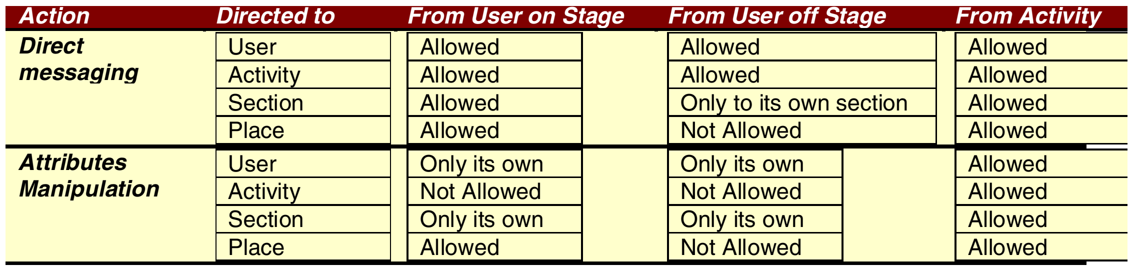
The places directory is available as well through the PlacesAdmin component of the server toolkit. It should be noted that in a distributed environment when there is more than one places server application the directory will give information only on the place that were created on the server you are connected to and not on the entire community.

3.4 Activities

Activities are server side applications that share the place with the users and define ways in which they can collaborate. The possibilities that are embodied in the 'activity' concept are vast. Collaborative web games, business a/v meetings, application sharing, monitor things that happen in the place and much more. The server toolkit provide a way for server application to register themselves as activities in a place and provides a way for them to communicate with the users in the place. Usually an activity will have its own propriety protocol with the client application that implements the specific tasks of that activity. An activity in a place is the most powerful member in terms of permissions. It can do basically anything: Monitor all the messages that are transferred in the place (even private messages between users), Has no limitation on awareness or communication (As oppose to regular users. See below the table of permissions in the place. Activity support is included in the server toolkit in the ActvityService component.

3.5 Permissions list

Figure 3-1 on page 29 lists the communication abilities of different members of the place. There are 3 main 'actors': User that is in the stage section, user that is on another section and an activity. For example, while a user on the stage is allowed to send messages to every other section in the place, a 'regular' user cannot.



Action	Directed to	From User on Stage	From User off Stage	From Activity
Direct messaging	User	Allowed	Allowed	Allowed
	Activity	Allowed	Allowed	Allowed
	Section	Allowed	Only to its own section	Allowed
	Place	Allowed	Not Allowed	Allowed
Attributes Manipulation	User	Only its own	Only its own	Allowed
	Activity	Not Allowed	Not Allowed	Allowed
	Section	Only its own	Only its own	Allowed
	Place	Allowed	Not Allowed	Allowed

Figure 3-1 Communications permissions list for place

It should be noted that as for receiving notifications on messages or on attribute changes there are no restrictions. Any member can receive messages that are sent to him either directly or not, and can watch the changes of attributes of any other member. The restrictions only apply to active operations like sending messages and changing attributes of members.

4

Installation and setup

This chapter describes how set up a Java Development Kit (JDK) so you can compile and test the sample programs described in this book. We also show you how to prepare the Sametime server to run server applications and how to register your Sametime server application as a service under Windows 2000.

Note: This chapter was written against an alpha version of the Sametime Community Server Toolkit where the Java classes and resources were delivered in a file names STCommServer25.jar. This file has been replaced with two files in the beta toolkit names stcommsrvrtk.jar and commres.jar

4.1 Installing the JDK

There are several places were you can download the Java Development Kit (JDK). If you don't have it already, we recommend downloading it from the IBM developerwork Internet site: http://www.ibm.com/developerworks and search for the JDK 1.3 (although the Toolkit supports JDK 1.1, it will run on the 1.3 platform). Once you download the JDK, install it. Try to compile and run one of the sample applications that come with the JDK to ensure it runs correctly before compiling any Sametime applications.

4.2 Preparing the Sametime server to run applications

First, let's set up the Sametime environment to handle server applications. You must tell Sametime that it is OK to accept IP connection from server applications on different machines. You can tell Sametime to accept application IP connections by doing the following:

1. On the server machine that is running Sametime, locate the sametime.ini file (generally you can find the sametime.ini file in C:\Sametime\sametime.ini depending on how you installed Sametime).
2. Open sametime.ini file with notepad
3. Find the [Config] section of the ini file
4. Add the following line in the [Config] section:

   ```
   VPS_TRUSTED_IPS= "trusted IP list separated by comma"
   ```

Alternatively, you can tell the Sametime to allow connections to it from any IP address. You can do so by following the previous steps 1 and 2, and do the following:

1. Add a section called [Debug] at the end of the ini file (if one doesn't already exist)
2. In the [Debug] section add the following line:

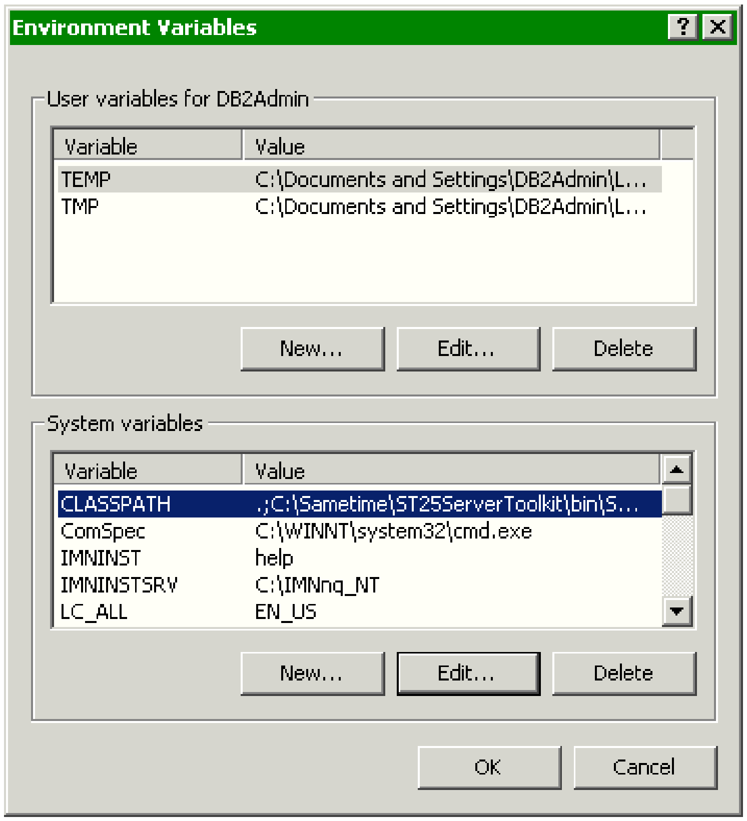
   ```
   VPS_BYPASS_TRUSTED_IPS=1
   ```

Now you are ready to proceed to step two of setting up your environment.

4.3 Setting up the classpath in the environment variable

In order for windows to find the required Sametime classes to run and compile java programs, you must add the toolkit jar file to the classpath variable. Follow these steps to do that:

1. Go to the **Start** menu
2. Choose **Settings -> Control Panel -> System** from the control panel
3. Choose the **Advanced** tab
4. Click the **Environment Variables** button, then you will get a window as in figure Appendix 4-1, "The Environment Variable window" on page 34



User variables for DB2Admin

Variable	Value
TEMP	C:\Documents and Settings\DB2Admin\L...
TMP	C:\Documents and Settings\DB2Admin\L...

System variables

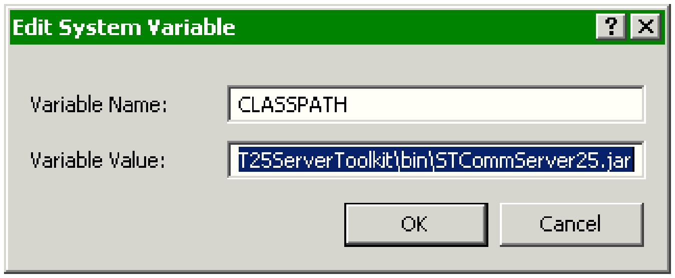
Variable	Value
CLASSPATH	.;C:\Sametime\ST25ServerToolkit\bin\S...
ComSpec	C:\WINNT\system32\cmd.exe
IMNINST	help
IMNINSTSRV	C:\IMNnq_NT
LC_ALL	EN_US

Figure 4-1 The Environment Variable window

5. In the **System Variables** section find a variable called **Classpath**
6. If you are unable to find the variable then click the New button. If you were able to find the classpath variable then select it by clicking on it, and then click the **Edit** button
7. In the variable value section, add a "." followed by a semicolon, then add the full path of the jar file name. For example, if the jar file was located in C:\Sametime\ST25ServerToolkit\bin directory, then the classpath should look like this:

   ```
   .;C:\Sametime\ST25ServerToolkit\bin\STCommServer25.jar
   ```

   You can also see that in Figure 4-2 on page 35
8. Finally click **OK** button



Variable Name: CLASSPATH

Variable Value: T25ServerToolkit\bin\STCommServer25.jar

Figure 4-2 The Edit System Variable Response window

4.3.1 Compiling the Program

Now you are ready o compile your first Sametime Server Toolkit application.

We will start out by showing how to do this in Chapter 5, “The hackers catcher sample” on page 37.

5

The hackers catcher sample

In this chapter we will discuss a simple application that comes with Sametime as part of the sample programs. The reason we chose this sample is because it reflects a practical problem in real life, yet it has a simple implementation of Sametime, which allows us to concentrate on the main features of the Toolkit without having to explore in depth the Java programming language.

Hackers Catcher Sample

The Hackers Catcher example is an application that notifies the console of every failed login attempt. For example, suppose that there is a hacker who is attempting to access the Sametime server. The hacker knows a UserID, and knows possibilities of the password. The hacker will attempt to login with several passwords to find out which one is the correct password. Most likely the first attempt will be unsuccessful (in reality unless the hacker has the correct password all attempts will be unsuccessful). The hacker will attempt to login several times unsuccessfully. However, you as a system administrator would like to monitor what kind of activities are taking place on your server, to be able to pursue preventative measures. Hackers catcher is a tool you can develop (or in this case use the already developed example).

The application will print to the screen the failed attempts. it will show the Login Name, the IP address of the machine, the type and the reason of the refusal to login.

In the next section, we will compile the application and run it. Then we will try to login in successfully and unsuccessfully to see what happens.

Compiling the Sample

First, find the file named HackerCatcher.java on your machine. Most likely you will find the file under the samples folder in the toolkit directory. Once you find the file launch the command prompt (In this chapter we will use the Java compiler from the command line). So at the command prompt, type the following:

```
C:\Javac HackersCatcher.java
```

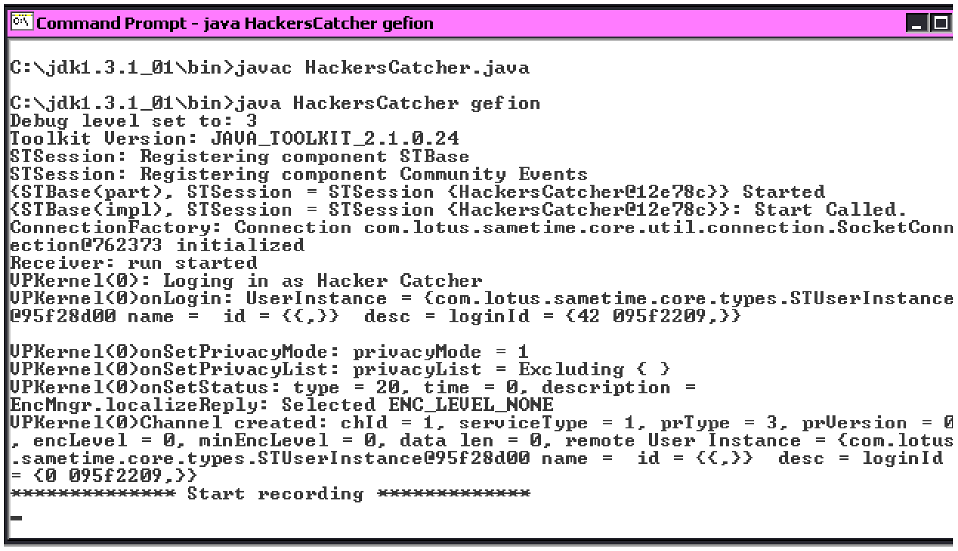
If the program compiles correctly no messages will appear and the command prompts starts a new line. like this:

```
C:\Javac HackersCatcher.java
C:\
```

The application takes a command line argument. The argument is the DNS name of the server that you would like to monitor. In our example we will run the application on the same machine, thus we will supply it with the full DNS name. The name of the server in this case is gefion.lotus.com. To run the program we can type the following:

```
C:\Javac HackersCatcher.java
C:\Java HackersCatcher gefion.lotus.com
```

You should have an output similar to the one in Appendix 5-1, “The output after invoking the HAckersCatcher program” on page 39



Command Prompt - java HackersCatcher gefion

```
C:\jdk1.3.1_01\bin>javac HackersCatcher.java

C:\jdk1.3.1_01\bin>java HackersCatcher gefion
Debug level set to: 3
Toolkit Version: JAVA_TOOLKIT_2.1.0.24
STSession: Registering component STBase
STSession: Registering component Community Events
<STBase(part), STSession = STSession (HackersCatcher@12e78c)> Started
<STBase(impl), STSession = STSession (HackersCatcher@12e78c)>: Start Called.
ConnectionFactory: Connection com.lotus.sametime.core.util.connection.SocketConn
ection@762373 initialized
Receiver: run started
VPKernel(0): Loging in as Hacker Catcher
VPKernel(0)onLogin: UserInstance = (com.lotus.sametime.core.types.STUserInstance
@95f28d00 name =  id = {(,)}  desc = loginId = (42 095f2209,)}

VPKernel(0)onSetPrivacyMode: privacyMode = 1
VPKernel(0)onSetPrivacyList: privacyList = Excluding { }
VPKernel(0)onSetStatus: type = 20, time = 0, description =
EncMngr.localizeReply: Selected ENC_LEVEL_NONE
VPKernel(0)Channel created: chId = 1, serviceType = 1, prType = 3, prVersion = 0
, encLevel = 0, minEncLevel = 0, data len = 0, remote User Instance = (com.lotus
.sametime.core.types.STUserInstance@95f28d00 name =  id = {(,)}  desc = loginId
= (0 095f2209,)}
************** Start recording **************
_
```

Figure 5-1 The output after invoking the HAckersCatcher program

Testing the application

To test the application, try to log on to the server from the Sametime connect client with the correct name and password. If you notice nothing will happen, because the application does not monitor correct logins. Alternatively, let's try to logout, then attempt to login with the same user id but different password. On the console you will get a notification of the failed login attempt. The message will indicate that the login has failed and produces a message similar to the following:

```
login failed: Name=user name, ip=gefion/255.255.255.255, type=1022,
reason=80000211
```

Try other different scenarios and observe the output to the screen.

In the next section, we will take a closer look into the application. We will walk through the code and provide an explanation where necessary.

5.1 Examining the code

This section has three main subsection. First we will discuss the packages imported by the example. Then we will discuss the main classes we are using within the example. Finally, we will discuss the methods and the mechanism in which the program functions.But first, let's take a quick glance at the HackersCatcher.java code:

```
import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.communityevents.*;

import java.net.InetAddress;

public class HackersCatcher implements
            LoginListener,
            UserLoginFailedListener,
            CommunityEventsServiceListener
{
    /**
      * The session object.
      */
     private STSession m_session;

      /**
      * The server application service.
      */
     private ServerAppService m_saService;

    /**
     * The Community events component
     */
    CommunityEventsService m_ceService;

    private void run(String serverName)
    {
        // init the sametime session and objects
        // First, we create a new session, that belongs uniquely to us.
          try
          {
             m_session = new STSession("" + this );
            m_saService = new STBase(m_session);
            m_ceService = new CommunityEventsComp(m_session);
          }
          catch (DuplicateObjectException e)
          {
            System.out.println("STSession or Components created twice.");
        }

      // start the session
      m_session.start();

      // Login to sametime
      m_saService.addLoginListener( this);short loginType =
STUserInstance.LT_SERVER_APP;
      m_saService.loginAsServerApp( serverName, loginType, "Hacker Catcher",
null);}

   public void loggedIn(LoginEvent event){
      m_ceService.addCommunityEventsServiceListener(this);
   }
    public void loggedOut(LoginEvent event){
m_ceService.removeCommunityEventsServiceListener(this);}

   public void serviceAvailable(CommunityEventsServiceEvent event)
   {
      System.out.println("************** Start recording *************");

      m_ceService.addUserLoginFailedListener(this);
   }

    public void serviceUnavailable(CommunityEventsServiceEvent event)
   {
      System.out.println("************** finish recording *************");

      m_ceService.removeLoginFailedListener(this);
   }

   public void userLoginFailed(UserLoginFailedEvent event)
   {
      String s = new String("login failed:");
      s += " Name=" + event.getLoginName();
      s += ", ip=" + event.getLoginIp();
      s += ", type=" + Integer.toHexString(event.getLoginType());
      s += ", reason=" + Integer.toHexString(event.getReason());

      System.out.println(s);
   }

   /**
    * Entry point of the application
    */
   public static void main(String[] args)
   {
      if ( args.length != 1 )
      {
            System.out.println("Usage: HackersCatcher serverName");
            System.exit(0);
        }

        new HackersCatcher().run(args[0]);
    }
}
```

5.1.1 The package section

As in any Java program, this one starts with importing packages. The first package it imports is the com.lotus.sametime.core.comparch. This package contains the STSession class. Its constructor is responsible for creating a Sametime session and adding it to the list of sessions. The other three Sametime packages provide a definition to the community API for server applications as in com.lotus.sametime.community, which includes STBase class and the ServerAppService interface. The com.lotus.sametime.communityevents however, contains the CommunityEventsComp class and the CommunityEventsService interface. A detailed discussion of declaring these services and instantiate the class will follow.

5.1.2 The class and service declaration section

In the following subsections, we will discuss the declaration of three major variables. These variables are, m_session, m_saService, and m_ceService. Those three create the skeleton of our application, therefore we created a special section to discuss each one.

STSession and m_session

Although the Java programming language allows the programmer to declare variables any where within the program, it is generally a good practice to declare the variables in the beginning of the code. This sample has generally followed suite except in specific areas. Let’s take a look at the first declared variable m_session. The variable m_session is of type STSession the heart and soul of the Sametime application. As we mentioned earlier this is the class responsible for starting a new Sametime session and adding it to the list of sessions. There are numerous useful methods associated with this class object. In this sample we will only use one of the methods offered by this class, the Start() method. This method starts the session.

ServerAppService and m_saservice

The second declaration is for the server application service, m_saService. This service will allow our program to define the community API. To implement this service we use the STBase class constructor. The STBase constructor takes STSession as its argument. Luckily, we have declared and instantiated an STSession as we discussed in the previous paragraph. We will use two methods offered to us by this service. These two methods are addLoginListner and loginAsServerApp. The first method adds our login listener to receive login/logout events. Think of it as a way to eavesdrop on the logins and logouts from the Sametime server. This service will allow our application to receive an loggedin event upon successful login. The second method, loginAsServerApp logs our application into Sametime as a server application, thus prompting a loggedin event (if it was successful in loging in).

CommunityEventService and m_saService

Thirdly on our most important variables list comes the community event service. m_ceService. This service provides the definition of the community events service API. You can use the CommunityEventsComp constructor to implement this service. We must also pass it STSession as an argument (see why STSession is the heart and soul of the Sametime application).

The Community Events Service allows our application to receive a variety of events generated by the community event server. Our application can receive these events by registering itself with the server for each family of events. One event of special interest to us is the userLoginFailed event. We need this event to sense a login failure. Remember, the hacker will attempt to login to our server several times (hopefully unsuccessfully, because the hacker doesn’t know which password is the correct one from the list of possibilities that possibilities available).In the next section we will explain what happens in more detail, for now we just register to receive the event. Before we register to receive the event, we need a mechanism to know whether we are receiving community event notification in real time. Of course there is a way to do that in Sametime, we use the addCommunityEventsServiceListner method, which we will discuss in more detail how it fits to the picture in the next section.

5.1.3 Building the application

Sametime programming is built around the event driven programming concepts. Through out our code, we will register for a service and trap for its events. Once the event is triggered, we (through the programming code) instruct our application to behave in a certain way. For example, if the server triggers a

LoginFailed event, we would like the application to print out a warning message to the screen. We can achieve this by inserting the appropriate code in the LoginFailed event. Let's take a closer look at the HackersCatcher code and experience event driven programming in action.

As in any stand alone Java program, HackersCatcher has a main functions. In this case, main() takes command line argument (the server's name), thus passing it to another method.

The method is called run(). It is part of the HackersCatcher class. This class is the main and only user defined class in our application. All of our operations take place inside this class. We will start following the execution of the program through logical steps. We will first look at the definition of the class. HackersCatcher implements three interfaces, they are the LoginListener, UserLoginFailedListener, and CommunityEventsServiceListener. Each of these interfaces documented extensively under the doc folder in the Server Toolkit installation.

The following is a line of code from the main function:

```
new HackersCatcher.run(arg[0]);
```

This line has two functions, the first is to create a new instance of the HackersCatcher class by using the key word new. Then the it invokes the run method within the new instance of the class. Now let's take a look at the HackersCatcher internals.

The run method within our class provides several functionality, one of which is to instantiate the STSession object as in the following line:

```
m_session = new STSession(""+this);
```

As we discussed before this line of code is responsible for signing up for the Sametime session object, but remember, just because we signed up for it doesn't mean it is live and running. Think of this as renting an apartment, you sign a contract but you haven't moved in yet. You officially rent the apartment, for all living purposes you do not occupy the space yet (In other words you still don't live there). One you rent a truck and move your belongings and start using the space then you are officially moved in. Same analogy goes to STSession. You declare the object (intention to sign a lease), you instantiate the object (sign the lease contract), and then use the start() method to jump start the session (get a truck and move your belongings and start living in the apartment).

Now that you have the session started, you can start adding new services to your application. One thing we will need is a login listener that will inform us when a login or a logout has occurred. We can do this by invoking the addLoginListener(LoginListener listener) method. We invoke this method as in the following line of code:

```
   m_saService.addLoginListner(this);
```

So now, our application is ready to receive login/logouts events. Once our server logs in, we will be able to receive a loggedin(LoginEvent event). Naturally, our next step will be to log in our application. A useful method we can use is loginAsServerApp method. However, one of the variables that this method requires is a loginType. There are several login types defined for us in the STUserInstance.

STUserInstance is a class included in the com.lotus.sametime.core.type package. It contains several useful methods and fields. Of importance to us are the fields that define login types. Here is a list of all the available login types:

```
LT_USER_JAVA_COMP: Login type for Java apples
LT_USER_JAVA_APP: Login type for Java applications
LT_SERVER_APP: Login type for server application
LT_MUX_APP: Login type for mux application
LT_LIGHT_CLIENT_USER: Login type for a light client.
```

For a detailed explanation of each, refer to the documentation included with the Java Server API under com.lotus.sametime.core.type -> STUserInstance. But for now, we will concentrate on the server application login type. We store this value in a local variable of type short and we will appropriately call it loginType. Then we can pass it as an argument to the loginAsServerApp method along with servername which we have from the command line argument, and use Hacker Catcher as the application name. Our code looks like this:

```
   m_saService.loginAsServerApp(servername, loginType, "Hacker Catcher",null);
```

Upon our successful login, an event is triggered. This event is called (as you probably guessed by now) loggedIn (remember we registered to receive logged/loggedout in the previous line of code).

After our application logged in successfully, we will start listening to events available in the community events package. However, before we start listening to such events, we must make sure that the community events service are up and running. Sometimes the community services are down due to some unforeseen problem like congested network for example. We can sign up to receive service available/unavailable events by using the following:

addCommunityServiceListener(CommunityEventServiceListener listener)

So if the community event service is available (which if the server is up and running then most likely available) we will get a serviceAvailable event.

Now, since we want to find out about failed logins to the server, we can sign up to receive a notification service which inform us of the failed login. To do this, we have another handy method within the COmmunityEventsService interface, and it is called addUserLoginFailedListner. Once we invoke this method, then we will be set up to receive an event called loginFailed. This events warns us to the failed login case. Within ti we can find all kind of useful information like the login name, the machine IP, and so on. Because of the availability of these parameters, we code this event to extract these parameters and print it to the screen, as in the following piece of code:

```
public void userLoginFailed(UserLoginFailedEvent event)
{
      String s = new String("login failed:");
      s += " Name=" + event.getLoginName();
      s += ", ip=" + event.getLoginIp();
      s += ", type=" + Integer.toHexString(event.getLoginType());
      s += ", reason=" + Integer.toHexString(event.getReason());
      System.out.println(s);
}
```

5.1.4 Cleaning up

One important part of our application is the cleaning upt.The best practice here is to counter every addMethod with a removeMethod. You can see that in the case of addCommunityEventsServiceListener in the loggedIn event. Naturally we should supply removeCommunityEventsServiceListener in the counter event which is in this case loggedOut. The code looks like this:

```
    public void loggedIn(LoginEvent event){
         m_ceService.addCommunityEventsServiceListener(this);
    }
    public void loggedOut(LoginEvent event){
         m_ceService.removeCommunityEventsServiceListener(this);
    }
```

Similarly we follow the same technic when we use addUserLoginFailedListener in the serviceAvailable event. We counter it with removeLoginFailedListener in the serviceUnavailable event.

5.2 Extending the application

The application now is in functioning stage where it will send a simple message to the console of the failed attempt. But suppose that you would like to add more useful functionality to the application. Instead of printing the message to the console, you would like the user to receive a message informing him/her of the failed attempt since the last successful log in. In the following section we will add more code to reflect this functionality.

5.2.1 Importing more package

First we will need to import four new packages into our application. These packages are:

```
com.lotus..sametime.lookup.*
com.lotus..sametime.im.*
com.lotus..sametime.constants.*
java.util.Hashtable
```

The lookup package will provide us with the needed mechanism to resolve the name of the user whose account is being hacked into. Since the login attempts were unsuccessful, we will not receive an STId for the user. All we will have is the name, even then it may not be a valid one. For now, we will concentrate on the user being a valid and resolvable.

The second package is the im for instant messaging. This service will provide us with the necessary means to send an instant message to user. Upon a successful logon, we will use the im services to send the user a warning message about the failed logins.

The constant package will allow us access to some classes that defines some critical constants which we will need in various APIs. The Hashtable package is not a part of the Toolkit, but a part of the standard JDK. The hashtable will be useful to store the users information to look for it later when the user logs in successful.

5.2.2 The scenario

Let's describe what we will do in plain english before we dive into the code. A user will attempt to logon but fails (for various reasons, but here we will assume an incorrect password). A userLoginFailed events get generated. In this event we will attempt to resolve the name. Successful resolution will result in a resolved event. In the resolved event we get the user info and store it in a STObject. We can use the getId method from within the STObject to create a hash table key and store the object in the hash table. Meanwhile, the application is listening to new "successful" logins. In such a case, a userLoggedIn events gets generated. In that event we will look for the STuserId in the hash table. If it is found, the application starts generating an IM and sends it to the user.

5.2.3 Implementing the Interface

There are three important interfaces that we will add to the original HackersCatcher application, they are the UserLoginListener, ResolveListener, and ImListner.

The UserLoginListener provides the application with the ability to listen for UseLogin events. Previously we implemented a similar listener that listens to the application logins (remember?). This interface will behave in the same manner except the events will be triggered by users loging in.

The ResolveListener interface is responsible for the ability to resolve names from the user name value, it returns an STUserObject. The reason we need this service is because when the a hacker attempts to login but never successfully. In order to send that user a message in the future, we must have a unique identifier for that user. What better identifier than the Sametime user id, which we will get by resolving the String type user name.

ImListener service will assemble an appropriate message and send it to the user whose account been attacked. In this case the message will be something along the lines "Failed login has been attempted". The message will be sent upon the first successful log in.

Next, we will explain how we added the necessary code to make this idea a reality.

5.2.4 Coding the application

In following our tradition, we will present the full code here before starting to describe it:

```
 import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.communityevents.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.im.*;
import com.lotus.sametime.core.constants.*;

import java.net.*;
import java.util.Hashtable;
```

```
public class HackersCatcher  implements
          LoginListener,
          UserLoginFailedListener,
          CommunityEventsServiceListener,
                        UserLoginListener,
                        ResolveListener,
                        ImListener



{

        /**
         * The session object.
        */
        private STSession m_session;

        /**
         * The server application service.
        */
        private ServerAppService m_saService;

        /**
        * The Lookup Component
        */
        private LookupComp m_LookupComp;
        /*The Resolver for the lookup*/
        private Resolver m_lookup;
        private ImComp m_imcomp;


        /**
        * The list of users we watch who have logged in unsuccefully.
        */
        private Hashtable m_watchedUsers = new Hashtable();
   /**
    * The Community events component
    */
   CommunityEventsService m_ceService;

   private void run(String serverName)
   {
      // init the sametime session and objects
      // First, we create a new session, that belongs uniquely to us.
        try
        {
          m_session = new STSession("" + this );
          m_saService = new STBase(m_session);
```

```
            m_ceService = new CommunityEventsComp(m_session);
                        //get the resolver to lookup names
                        m_LookupComp = new LookupComp(m_session);
                        m_lookup = m_LookupComp.createResolver(true, false, true,
false);
                        //get the IM
                        m_imcomp = new ImComp(m_session);


          }
          catch (DuplicateObjectException e)
          {
            System.out.println("STSession or Components created twice.");
          }

        // start the session
        m_session.start();

        // Login to sametime
        m_saService.addLoginListener( this);
                  short loginType = STUserInstance.LT_SERVER_APP;
        m_saService.loginAsServerApp( serverName, loginType, "Hacker Catcher",
null);
          }

    public void loggedIn(LoginEvent event){
        m_ceService.addCommunityEventsServiceListener(this);
                  //Start Listening to resolve events
                  m_lookup.addResolveListener(this);

    }
          public void loggedOut(LoginEvent event){
                  m_ceService.removeCommunityEventsServiceListener(this);
                  m_lookup.removeResolveListener(this);

          }

    public void serviceAvailable(CommunityEventsServiceEvent event)
    {
        System.out.println("************** Start recording *************");

        m_ceService.addUserLoginFailedListener(this);
                  m_ceService.addUserLoginListener(this);

    }

          public void serviceUnavailable(CommunityEventsServiceEvent event)
    {
        System.out.println("************** finish recording *************");
```

```
      m_ceService.removeLoginFailedListener(this);
               m_ceService.removeUserLoginListener(this);
   }

   public void userLoginFailed(UserLoginFailedEvent event)
   {
      String s = new String("login failed:");
      s += " Name=" + event.getLoginName();
      s += ", ip=" + event.getLoginIp();
      s += ", type=" + Integer.toHexString(event.getLoginType());
      s += ", reason=" + Integer.toHexString(event.getReason());
               m_lookup.resolve(event.getLoginName());
      System.out.println(s);

   }
        public void userLoggedOut(UserLoginEvent userLoginEvent) {
        }

        public void userLoggedIn(UserLoginEvent userLoginEvent) {
                // Check if we are interested in this user.
                STUser user = userLoginEvent.getUserInstance();
                Object o = m_watchedUsers.remove(user.getId());
                if (o != null)
                {
                    STObject m_STObject = ((STObject)o);
                    // Finally we can send the message.
                    Im im = m_imcomp.createIm(user, EncLevel.ENC_LEVEL_ALL, 1);
                    im.addImListener(this);
                    im.open();

               }
        }
        public void resolved(com.lotus.sametime.lookup.ResolveEvent event) {
            STObject m_STObj;
            m_STObj = event.getResolved();
            m_watchedUsers.put(m_STObj.getId(), m_STObj);
        }

        public void resolveFailed(com.lotus.sametime.lookup.ResolveEvent
resolveEvent) {
        }

        public void resolveConflict(com.lotus.sametime.lookup.ResolveEvent
resolveEvent) {
        }

        public void imOpened(ImEvent event)
        {
            // Send the message and leave asap.
            event.getIm().sendText(true, "a User Attempted to login with your
ID");
            event.getIm().close(0);
        }
        public void openImFailed(com.lotus.sametime.im.ImEvent event) {
            System.out.println("SA HANDLER: Couldn't open IM " +
                                  event.getReason());

        }


   /**
    * Entry point of the application
    */
   public static void main(String[] args)
   {
      if ( args.length != 1 )
      {
                  //new HackersCatcher().run("gefion");
         System.out.println("Usage: HackersCatcher serverName");
         System.exit(0);
      }

      new HackersCatcher().run(args[0]);
   }


}
```

5.2.5 Explaining the code

As always the application begins by importing the packages, then it implements the three interfaces in the class definition as in the following:

```
public class HackersCatcher  implements  LoginListener,
UserLoginFailedListener,CommunityEventsServiceListener,
                        UserLoginListener,
                        ResolveListener,
                        ImListener
```

Next in the application comes the declaration section. There are four new global declarations introduced to HackersCatcher class. To resolve the name we will need the LookupComp class which implements the lookup service. We also declare a Resolver class, which provides the method to resolve the user name.

The application also contains a declaration for the ImComp class, which is the component responsible for implementing the instant messaging service. Later we will declare an IM class to use in sending the instant message, but not until we find the user (once the user logs in).

FInally, the application utilizes the Hashtable class provided by the JVM. There are two important methods we will use within this class the put and move. In the next few sections we will explain how these methods integrate into our code.

Instantiating the classes

Since the hash table is independent of any other objects within our application, we will just instantiate it as it's declared by using:

```
 new Hashtable()
```

In order to use the resolver, first we must instantiate the lookup component by using the class constructor LookupComp(STSession), we will pass it the m_session. Once instantiated, we will have access to its createResolver method, which returns a resolver object. This method takes four boolean variables, it is declared as the following:

```
public Resolver createResolver(boolean onlyUnique,
                               boolean exhaustiveLookup,
                               boolean resolveUsers,
                               boolean resolveGroups)
```

The onlyUnique variable indicates whether the resolver object should resolve only if there is a unique match for the name. For simplicity's sake, we will make this value true. The exhaustiveLookup tells the resolver object whether to stop at the first successful resolve or continue on to other directories even after the first match, we will set it to false. The resolveUsers flags whether the object should resolve for the names of individual users where resolveGroup indicates whether the created resolver should resolve the names of groups. Finally, we use the ImComp to construct a new Instant Messaging component and associate it with the Sametime session.

Adding the listeners

As soon as the application log in successfully, it should begin to listen for resolver events. There are three events that get generated by the resolve, these events are resolveConflict, resolved, and resolveFailed. The first event, resolveConflict indicates that the resolve request has generated multiple matches, however, in our case we will not get this event simply because in the createResolver method, we indicated that the resolver should resolve successfully by resolving uniquely (first passed value was true from the previous section). The other two events are generated upon successful resolve or a failed resolve respectively.

In the serviceAvailable event we add a userLoginListener by using the method addUserLoginListener. This listener is similar to the one added immediately after the start of the session (addLoginListner) except the newly added listener waits for user logins, where the first listener waits until the server application logs in. In this application we have a need for both functionality. The new listener tracks two events userLoggedin and userLoggedOut. Another type of listener we need is the instant messaging listener (addImListener). In order to follow the flow, we will provide a more through discussion of this method later in this chapter.

Assembling the application

The first logical indication to mobilize the application is once a userLoginFailed event is triggered. This event has access to the user's name whose attempt to login is failing. Therefore, we get the name by using event.getLoginName() (again) and pass that value to the resolve method. The call will finally look like the following:

```
m_lookup.resolve(event.getLoginname());
```

Luckily (since we signed up for a resolve listener earlier), we will get an either resolveFailed or resolved event. In the resolved event, the application has access to the Sametime object STObject which has methods that can access the STId. We retrieve the Sametime object by invoking getResolved within the event which returns an STObject. STObject is an interface for base Sametime community entities. It provides us with methods to retrieve the STId and the name of the objects. Since a unique STId is available to us which is assigned to each user, we can use it as the key value to store the object in the hash table. To achieve this objective we use put method within the Hashtable object:

```
m_watchedUsers.put(m_STObj.getId(), m_STObj);
```

Storing the object in the hashs tabe is only half the battle. We now need to wait for the user to login, to deliver the message. A good place to look for our intended user is the userLoginEvent. In that event we can wait until the intended user log is. Everytime we get a userLogin event we check the hash table to see if this is the user we ned to contact. A simple test would be to get the STId of the logged in user. Then we attempt to remove that ID value form the hash table by using remove(user.getId()) which returns a universal Java object if it was successful and null if it wasn't. Therefore, we check the object returned for null, and if it is not then (Bingo) the intended user is logged in. Once the user is logged in the last part of the battle is in order. Now we can compose the Instant Message in order to send it. A handy method we may use offered by the previously declared IM component m_imcomp is createIm. We will pass this method the STUser object which we already have from the resolve event. The method also requires a value to indicate the encryption level of type EncLevel. We have access to an array of encryption level in the constants package under

EncLevel. For simplicity we will choose the any type of encryption level ENC_LEVEL_ALL. Finally the method requires an int value indicationg the typ of the IM and we will use 1 to serve our purpose. The final picture will look like the following:

```
Im im = m_imcomp.createIm(user, EncLevel.ENC_LEVEL_ALL, 1);
```

Next we will add an IM listener which will warn the application when an IM session has been opened or closed. It also waits for text and data received events, and it also looks for an openImFailed event. We are only concerned about the imOpened and openImFailed events for now. Because the next step after composing the IM and adding the listener is to open the IM session with the open() method. This is how things will look:

```
im.addImListener(this);
im.open();
```

Now there are two possiblities (everythin remaining constant); either the IM session open failed or succeeded. If opening the session fails we print a message to the console with the reason of the failure. The code to do this can be inserted into the openImFailed event. If opening the session was successful then we will send the user a message and close the session immedietly to ensure that the user will not reply. To send the IM from the imOpened event, we first need to get the IM object which can be achieved by using getIm method within the event. We can also in the same line of code send the IM be using sendText method provided by the IM object. This method takes two arguments. The first a boolean indicating whether the message should be encrypted or not, the second is the string to be sent. Upon sending the message we must use the close method in the IM object which will close the IM session. We need to close the session because if we don’t then the user will be able send messages to our application thinking that s/he is chating with a human administrator. And since our application is not trapping for messages to be sent back, nothing will happen. Therefore, in order not to deceive the user into thinking that a response back is in the work for the comments, we clsoe the IM. If the user attempts to send messages back to the server, an error message gets generated on the users machine indicating that the IM session has ended and no more messages can be sent. This is how your code should look like:

```
event.getIm().sendText(true, "a User Attempted to login with your ID");
event.getIm().close(0);
```

5.2.6 Knowledge exercise

There are several other events that we mentioned but didn't act upon for a purpose. Some of these events are imClosed, textReceived and so on. A good exercise would be to try to code some of these events to see how such events behave. Coding these events on your own will give you a better feel for the application and the behavior of its APIs.

6

The places logger sample

In this section we will switch gears. Our new quest will concentrate on listening to services performed by the server rather than a user. In the HackersCatcher application, we listened to user activities, user logins. user log outs, user login failed to name a few. In the next section our application is more advance. It will listen to events performed by the server. Before we dive into the code, we will provide a refresher from a previous chapter. Then we will provide the code which is very similar to the PlacesLogger application in the samples except it does not have the nice table. Instead it logs information into the console.

Note: The code we examine in this chapter is a drilled-down version of the PlacesLogger sample provided with the toolkit. Most of the user interface code has been removed to allow us to concentrate on the Sametime code.

6.1 Places and Activities

Sametime is very unique in how it handles services. Each service performed by the server has an application server that is responsible for delivering it. For example, in order for users to create meetings there are several application server take place in the background. Two of these applications are the Place Administrator and the Activity Administrator. The client asks for a meeting place. But the place is worthless if doesn’t include activities to conduct the meeting. Currently, these activities are audio, video, whiteboarding and chat capabilities. Let’s assume that a user needs a place with all of these capabilities. The client request a place. Then it will also request activities to associated with this place, but the activities are provided by a different server task than the task that provides the place. Now suppose that we want to create an application that will provide a list of places on the server as soon as they are created. Our application should also provide a list of the available tools in the place. In order to provide a list of the places we can provide the application with a place listener. But to get the tools within the place we will need an activity listener. This listener will receive an activity requested event which will track the type of activities included in the place.

Let's take a look at the code and see how the theory is put to practice.

6.2 PlacesLogger Java code

Here is the code:

```
import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.placessa.*;
import com.lotus.sametime.core.types.*;

/**
 * A sample of the places sa toolkit. Monitors the places that are created and
 * logs their operations.
 */
public class PlacesLogger implements LoginListener, ActivityServiceListener,
PlaceListener
{
    //
    // Activity types that a place may contain
    //
    public final static int CHAT_ACTIVITY         = 0x9106;
    public final static int AUDIO_ACTIVITY        = 0x9103;
    public final static int VIDEO_ACTIVITY        = 0x9104;
    public final static int SCREEN_SHARE_ACTIVITY = 0x9102;
    public final static int WHITEBOARD_ACTIVITY   = 0x9101;


    /**
     * The activity type we support.
     */
    static final int ACTIVITY_TYPE = 0x0010;

    /**
     * The type of places we log.
     */
    static final int PLACE_TYPE = 0;
    // Members.
    /**
     * The session object.
     */
    private STSession m_session;

    /**
     * The places admin service.
     */
    private PlacesAdminService m_adminService;

    /**
     * The activity service.
     */
    private ActivityService m_activityService;


    /**
     * Constructor.
     *
     * @param hostName The name of the server to connect to.
     */
    public PlacesLogger(String hostName)
    {
        // Create and load the session of components.
        try
        {
            m_session = new STSession("PlacesLogger");

            new STBase(m_session);
            new PlacesAdminComp(m_session);
            new ActivityComp(m_session);

            m_session.start();
        }
        catch(DuplicateObjectException e)
        {
            e.printStackTrace();
            exit();
        }

        // Get a reference to the needed components.
        m_adminService
            =
(PlacesAdminService)m_session.getCompApi(PlacesAdminService.COMP_NAME);
       //Add a place listner, the events are defined in class AdminHandler
        m_adminService.addPlacesAdminListener(new AdminHandler());
        m_activityService
            = (ActivityService)m_session.getCompApi(ActivityService.COMP_NAME);
      //Provide activity listner and wait for activityRequested
        m_activityService.addActivityServiceListener(this);

        login(hostName);

    }

    /**
     * Login to the sametime server as a server application.
     *
     * @param hostName The name of the host.
     */
    void login(String hostName)
    {
        ServerAppService saService =
            (ServerAppService)m_session.getCompApi(ServerAppService.COMP_NAME);

        short loginType = STUserInstance.LT_SERVER_APP;
        int[] supportedServices = { ACTIVITY_TYPE };

        saService.addLoginListener(this);
        saService.loginAsServerApp(hostName, loginType, "PlacesLogger",
                                   supportedServices);
    }

    /**
     * Terminate the application.
     */
    void exit()
    {
        m_session.stop();
        m_session.unloadSession();
        System.exit(0);
    }

    //
    // Places Activity Listener
    //
    /**
     * A request for an activity as come from a place.
     *
     * @param event The event object.
     * @see PlacesActivityEvent#getManagedActivity
     * @see PlacesActivityEvent#getPlace
     */
    public void activityRequested(ActivityServiceEvent event)
    {
        //An activity has been requested
          MyActivity activity;
          // A place was created and our activity is requested.
          Place place = event.getPlace();
        //store MyActivity in a variable for easier refrence
          activity = event.getMyActivity();
          System.out.println("Place: " + place.getName());
        //Accept the activity request
          m_activityService.acceptActivity(activity, null);
        //inside the place add a listner to track when it gets destroyed
        place.addPlaceListener(this);
          //new PlaceHandler(event.getMyActivity(), m_frame);
    }

    //
    // Login Listener.
    //

     /**
     * Indicates that the PlacesLogger App logged in successfully to the
Sametime community.
     *
     * @param event The event object.
     * @see LoginEvent#getCommunity
     */
    public void loggedIn(LoginEvent event)
    {
        System.out.println("Logger: LoggedIn");
    }

    /**
     * Indicates that we were successfully logged out of the Sametime
community, or a login
     * request was refused.
     *
     * @param event The event object.
     * @see LoginEvent#getReason
     * @see LoginEvent#getCommunity
     */
    public void loggedOut(LoginEvent event)
    {
        System.out.println("Logger: LoggedOut" + event.getReason());
        exit();
    }



    public void activityAdded(com.lotus.sametime.placessa.PlaceEvent
placeEvent) {
```

```
            int type = placeEvent.getActivityType();
            switch (type)
            {
                case CHAT_ACTIVITY:
                     System.out.println("Place: " +
placeEvent.getPlace().getName()+" has Chat Capability");
                     break;
                case AUDIO_ACTIVITY:
                     System.out.println("Place: " +
placeEvent.getPlace().getName()+" has Audio Capability");
                     break;
                case VIDEO_ACTIVITY:
                     System.out.println("Place: " +
placeEvent.getPlace().getName()+" has Video Capability");
                     break;
                case SCREEN_SHARE_ACTIVITY:
                     System.out.println("Place: " +
placeEvent.getPlace().getName()+" has Screen Sharing Capability");
                     break;
                case WHITEBOARD_ACTIVITY:
                     System.out.println("Place: " +
placeEvent.getPlace().getName()+" has WhiteBoard Capability");
                     break;
            }
        }


        //
        // Admin Listener.
        //

        /**
         * Handles places admin events.
         */
        class AdminHandler extends PlacesAdminAdapter
        {
            /**
            * The Places service is available.
            *
            * @param event The event object.
            */
            public void serviceAvailable(PlacesAdminEvent event)
            {
                // Set our activity as a default activity for the place type we
                // are interested in.
                System.out.println("AdminService available");
                m_adminService.setDefaultActivity(PLACE_TYPE, ACTIVITY_TYPE, null);
            }
            /**
            * The Places server is unavailable.
            *
            * @param event The event object.
            */
            public void serviceUnavailable(PlacesAdminEvent event)
            {
                System.out.println("AdminService un available");
                exit();
            }
            /**
            * The 'Set default activity' operation succeded.
            *
            * @param event The event object.
            * @see PlacesAdminEvent#getPlaceType
            * @see PlacesAdminEvent#getActivityType
            * @see PlacesAdminEvent#getActivityData
            */
            public void defaultActivitySet(PlacesAdminEvent event)
            {
                System.out.println("Default activity set");
            }

            /**
            * The 'Set default activity' operation failed.
            *
            * @param event The event object.
            * @see PlacesAdminEvent#getPlaceType
            * @see PlacesAdminEvent#getActivityType
            * @see PlacesAdminEvent#getActivityData
            * @see PlacesAdminEvent#getReason
            */
            public void setDefaultActivityFailed(PlacesAdminEvent event)
            {
                System.out.println("Set default activity failed");
                exit();
            }
        }
        /**
         * Application entry point.
         */
        public static void main(String[] args)
        {
            if (args.length != 1)
            {
                System.err.println("Usage: PlacesLogger [serverName]");
                System.exit(0);
            }
            new PlacesLogger(args[0]);
        }
    }
```

In describing this code we will follow the same format that we adopted in describing previous applications. Our agenda is to introduce the new packages, then the new implementations, finally we will discuss the classes and methods in detail.

6.3 The new package

The only package that you may not be familiar with from previous discussion is the com.lotus.sametime.placessa package. This package includes the APIs responsible for places and activities. It presents the capability of monitoring places and their associated activities. Before we move further, we must make a distinction between the Places Admin Service and the Places Activity Service.

Places Admin Service

The Places Admin Service provides the developer with the ability of administering the Places server application. Within this service the developer can create persistent places and control the characteristic and behavior of the created places. For example, when a developer creates a place, s//he can define the capacity of each section in a place and the default activities in it. Sections are a part of a place, if a user logs into a place, then the user will also be a part of a section and will also use an activity in that place. You can think of this concept as you would when you build a house. A house (place) is divided into rooms (sections) when you are in the house you will be in one of the rooms within the house. The house has utilities (activities) electricity, water, heat, cable and so on. A resident in the house will use some or all of these activities and in the future can even remove some of the utilities because that particular service (activity) is not useful anymore. For instance, the resident can remove cable because watching TV is not a priority any more.

Places Activity Service

An activity in an application that runs in a place and is shared by all of its members. Communications between the activity provider and the user can be implemented in a propriety protocol. An activity provider in a place is a super user. It can perform operations that regular users can not. For example, an activity provider can monitor all messages in a place, it can control how users are located in a place, and even changing the capacity of a section.

Normally when a place gets created it includes a default activity that have been associated with this particular place type. For example, N-way chat has the Chat activity as the default activity for it, therefore upon creating the place, the Chat activity is immediately created. But what happens if the users decided that they need a new activity, whiteboarding for example, how can they add it? The Toolkit provides a method that establish just that, adding a new activity. This method is defined in the Place class in the toolkit. It is Place.addActivity(int, byte[]).

In order for the activity provider application to receive such requests (addActivity()) it must register as an Activity Service Listener to the activity service. Whenever a request is issued for this service an activityRequested event will be triggered. The event has two methods, one returns the Place and the second method returns MyActivity object which represents the activity in the place. The activity provider application must respond by either acceptActivity() or declineActivity() within this event.

The Activity Service API contains objects that correspond to the virtual entities in a Sametime place. These objects are Place, Section, Activity, UserInPlace and MyActivity. These objects are defined to be place members, thus share common operations and events. Each of these objects define a listener which it can use to receive its own events. If an application is interested in one of these objects it should implement the listener interface associated with such object. Then the application add the listener method. For example, if the application is interested in the Section object then it would implement the SectionListener. Then the application signs up for the listener by using the method Section.addSectionLister(). Then the application will be able to trap for the various section events. One final note on the Activity Server Application, the application must declare a unique id number for the service it provides.

If you feel that this subject is still a bit unclear, don't worry. Concepts will be easier to understand as we start discussing the actual code in the next few pages.

6.4 Implementing the interface

The PlacesLogger class implements the LoginListner interface. It is the same interface we implemented in the HackersCatcher class. The interface is fully defined in the com.lotus.sametime.community package.

The class also implements the ActivityServiceListener interface. This interface is responsible for tracking the activityRequested event mentioned earlier. It provides the class to track the place's activity.

Finally, the class implements the PlaceListener. This is the listener interface that will track the events within the Place object. It will receive specific place events.

We can also implement the PlaceAdminListener interface here. However, we choose no to. Instead, we create another class AdminHandler that extends the PlacesAdminAdapter. When a class implements an interface, the Java compiler assumes that the developer will only access the provided methods. In fact when we tried to use the implements the compiler kept asking for all the events to be in the class. However, when we used extend, the compiler allowed us to use which ever methods we wanted and did not dictate that we must declare all the rest. As any Java developer will gladly tell you that interfaces can have subinterface, just as classes can have subclasses. The new class inherits all the implemented methods of the superinterface. Typically in Java you can even add new constants and methods and such. We will not do that here, we will only use some of the events provided to us by the interface. We only felt that we should mention this to give you a feel of the different options that are presented to you by the Toolkit.

To learn more about extend and implement we suggest that you refer to the JDK documentation or one of the books written about the subject. There are ample resources dedicated to this subject alone.

6.5 The variables and constants

The PlacesLogger class begins (as in most classes) with defining constants and variables. The first set of constants are the activity types. Each of these values represent an integer activity type. One constant for each of the known activities. We also define a new unique activity type that we support (ACTIVITY_TYPE = 0x0010). This is a unique number that has not been registered for any other service. Our application will use this value when it logs in to the Sametime server as a server application. This is the service unique id, in which our application will be identified as a service activity application.

In the variable section there are two new variables that weren't introduced in the HackersCatcher application. These two variables are m_adminService and m_activityService, and are of type PlacesAdminService and ActivityService respectively. The first variable includes several methods that allows us to configure the state and behavior of the virtual places created by a client. We are interested in one important method, the addPlacesAdminListener() method. The ActivityService class provides four methods, acceptActivity(), declineActivity(), addActivityServiceListener(), and removeActivityServiceListener(). We will use an instance of this service to add the activity listener and to accept activity request.

6.6 Understanding the code

We will provide explanation of the code here in chunk style. We will group bits of code under a different sub section name. The sub section will bare the name of the method. We are adopting this style for now to allow you to refer to each section with ease in the future.

The PlacesLogger constructor

The constructor instantiates the m_session just as we did in HackersCatcher or any Sametime application for that matter. Then we use new STBase() to construct a new community component. Unlike HackersCatcher however, we don't immediately get a reference to store in an ServerAppService. Instead, we wait to get the reference later. You may be wondering as to why we are doing this. There are two reasons; the first reason is that we did not declare a field for the PlacesLogger of that type, we choose to declare a variable of that type in one of the methods. The second reason is to illustrate a handy way of getting the reference later in the code. Now is a good time to discuss this new way. In reality we can easily declare a variable of type ServerAppService and then capture the reference in that variable. But you can achieve the same objective by calling STSession.getCompApi(String). This method returns type STCompApi which is the interface that represents a Sametime component's external API. Many of the services interface extend the STCompApi (remember the previous discussion about extend?). Once we get the return we can type cast it to the appropriate type we need as we will see later and viola! we get a reference to the intended APIs.

Tip: getCompApi() requires a string to be passed to it. Each service has a field called COMP_NAME and you can pass that field along

Once we construct a new community component, we also construct a new PalcesAdminComp and an ActivityComp, then we start the session.

Now we would like to get a reference (or access) to the addPlacesAdminListner but we know that we only have constructed a component but we don't have a reference to it so we will get a reference to it by invoking the getCompApi method the same way we described earlier. so the final picture will be as follows:

```
m_adminService = (PlacesAdminService) m_session.getCompApi
(PlacesAdminService.COMP_NAME);
```

Immediately after this line executes we get a reference to the PlacesAdminService components ant the APIs for it will be loaded in memory.

Next we add the listener for PlacesAdminService like this:

```
    m_adminService.addPlacesAdminListener(new AdminHandler())
```

You are probably wondering why are creating a new instance of class AdminHandler here and not use the key word this as we did in all the other times we used addXXXListener method. The answer is simple, we did not implement the PlacesAdminListener in the PlacesLogger class, instead we decided to create a different class that "extends" the PlacesAdminAdapter (remember our discussion about extend and implement?) All what this does differently than using this is it routes the events to that particular instant of AdminHandler class. It will handle any event we get appropriately, thus it got its name AdminHandler.

Next we ask for a reference to the activity service the same way we asked for the PlacesAdminServices. We add an activity listener but we will use the key word this. Again we can just as easily create a class that extends the ActivityAdapter and call it ActivityHandler to be consistent, create a new instance and pass it to the addActivityServiceListener() method.

The last thing we will have to do is to log our application to the Sametime server. We do that in a local method we declared called login. The next section discusses that method.

Logging in using login()

Here we will log in to the Sametime server. As we found out in the HackersCatcher example we will need access to the server application service APIs. We can get a reference to the APIs by using the STSession.getCompApi(string) and type cast the return into ServerAppService. We can use the saService to add alogin listener so we would be notified of a successful login once the next line of code executes.

The next line is to use the loginAssServerApp() method to login, just as we did in the HackersCatcher application with one exception. The exception is that the last parameter will not be null as in the HackersCatcher application. Instead, we will pass it an integer array with the only one member and that is the ACTIVITY_TYPE parameter, which will serve as the unique service type id that our application will be addressed as.

activityRequested()

The addActivityServiceListner() listens for this method. Once any place requests an activity, this event gets fired. Any application must respond here if it wishes to accept the activity requested or deny the request. In this method we also have access to the Place and MyActivity objects. We will get each by using getPlace() and getMyActivity() respectively. We will need a reference to these objects in order accept the activity using acceptActivity(), and to add a place listener to the place requesting the activity. The difference between adding a place listener here than adding a place AdminListener is that the later listens to events that normally happens on the outside of the place. Let's use the place and house analogy that we used earlier. Suppose that you are in town where they are building houses. You can monitor when a house is being built or destroyed. You can also see if the utility cables are being extended into the house and so forth. The AddPlacesAdminListner() behaves in the same way. It will only be able to monitor events associated with the outside of the place such as creating a place, destroying a place, and so on. If someone asks you to come an stay at their house while they are on vacation and take care of it. They also want you to make sure that service workers come in a perform the intended service. Then what you would need is a specific address identifying their house (getPlace()). Once you know where the house is, you will go inside it and monitor all the activities that may take place. But you are only able to provide the information on this one house, to provide the information about a different house, we need the address and a different person. The same logic applies to addPalceListener(), the listener will only monitor the activities inside the place. There is a list of all the events it looks for, one of particular interest to us is activityAdded().

The activityAdded() method

Once we receive an activity added event then we would like to know what it is and print the result to the screen. To get the event's id we can use getActivityType() which returns the integer type id. But in order to print things to the screen that people can understand we must convert this id into a more appropriate term for human communication. Remember those constants we defined in the beginning of the application, each one of these constants refers to one specific id of the five activities that we know of (chat, audio, video, screen sharing, and white board). We looked through each activity added and process it through a switch statement. Then print each sentence that refers to the appropriate activity.

The placeLeft()

This event gets generated as soon as the last user leaves the place. Sametime destroys the place soon after it was left by everyone. The server waits for few moments and then destroy the place. And finally the application prints the message to the console that the place has been destroyed.

Wrapping up

We realize that some of the code we implemented did not impact the application much, but we feel that showing you the code may help enhance your understanding. We encourage you to take this code modify it and study its behavior.

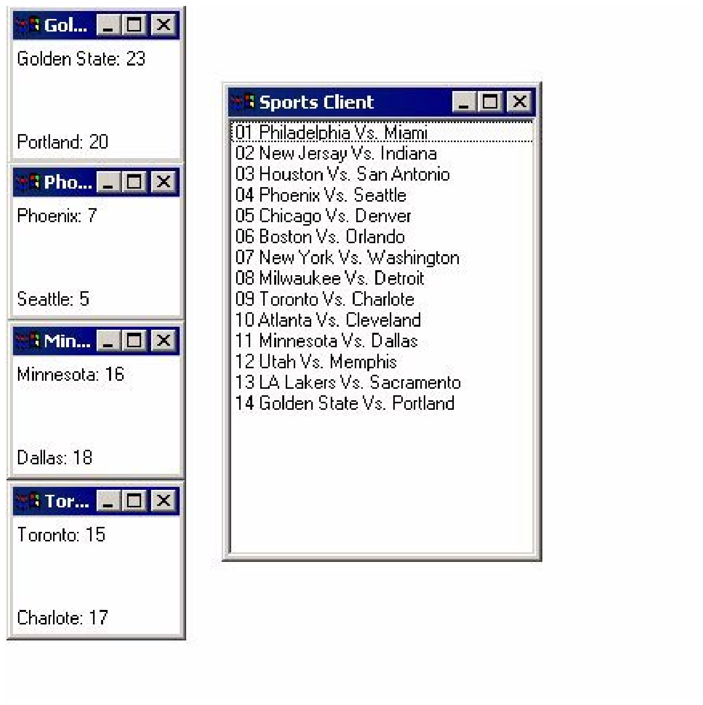
7

Developing a Sametime service

Imagine the next picture: You are sitting near your desktop, trying to complete this mission critical piece of code, but you just can't concentrate, your favorite team is in the middle of a playoff game, you open your browser in sportsonline.com, and start pushing the refresh button again and again. Sound familiar? well, in the chapter we show the SportsUpdater, a sample application which solve the "I can't wait anymore" problem.

7.1 SportsUpdater overview

Here is a snapshot of the SportsUpdater client while working.



Golden State: 23
Portland: 20

Phoenix: 7
Seattle: 5

Minnesota: 16
Dallas: 18

Toronto: 15
Charlote: 17

01 Philadelphia Vs. Miami
02 New Jersay Vs. Indiana
03 Houston Vs. San Antonio
04 Phoenix Vs. Seattle
05 Chicago Vs. Denver
06 Boston Vs. Orlando
07 New York Vs. Washington
08 Milwaukee Vs. Detroit
09 Toronto Vs. Charlote
10 Atlanta Vs. Cleveland
11 Minnesota Vs. Dallas
12 Utah Vs. Memphis
13 LA Lakers Vs. Sacramento
14 Golden State Vs. Portland

Figure 7-1 SportsUpdater in action

When the application starts, it pops up the Sports Client window, in this window you can see the list of all available matches, double click on an list item immediately opens the match window, which displays the current score.

7.2 Design

There are five classes and one interface in the sample:

- Class Match - represents one match, has a unique match ID which identify it. It also manage a list of MatchListeners and update them when there is a change in the match. Used both by server application and the client application.
- Interface MatchListener - declare the match listener interface.

- Class SportsUpdater - The main server application class, it manages the matches list and keep them up to date. It also responsible to login to Sametime and wait for clients request, when a client open a channel it creates a UserHandler for this client and adds it to his UserHandlers list.
- Class UserHandler - Handles one user request, when the user (client) first creates a channel to the SportsUpdater, the SportsUpdater creates a UserHandler for this user. The UserHandler holds an open channel to the user, and it listen to the user requests, when a user subscribe to a match the UserHandler adds itself as a MatchListener to this match. When a Match send notification about an update the UserHandler send it to its client.
- Class SportsClient - The main client application class. After logged in to Sametime it opens a channel to the server application, as a respond the server sends a match snapshot message on this channel. The client display all the matches in a window, and when the user double click a match, it sends a subscribe message on the channel and opens a window to display the match status.
- Class MatchFrame - The match status window.

7.2.1 Class diagrams

Server Application Class Diagram

Here are the main relationships between the server application classes.

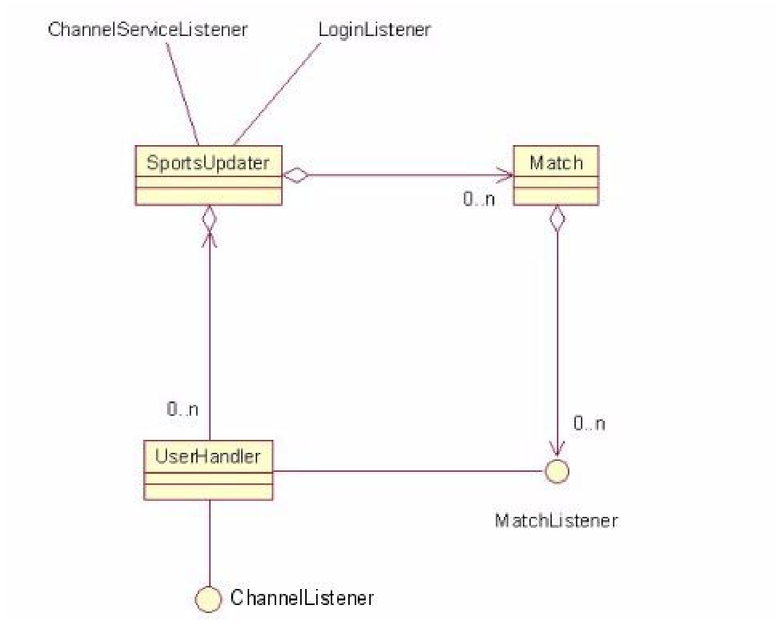


Figure 7-2 Server Class Diagram

Walkthrough: SpoortsUpdater implements the login listener so it can gets notification from the Sametime server when it's logged in, it also implements the ChannelServiceListner which elerts him whenever a user opens a channel to it. Sports updater handles two lists, a matches list and a user handlers list.

A Match holds a list of MatchListener so it can update them when the status of the Match has changed.

UserHandler implements the MatchListener interface, it also implements the ChannelListener so it can get messages from its client.

Client Class Diagram

Here are the main relationships between the client application classes.

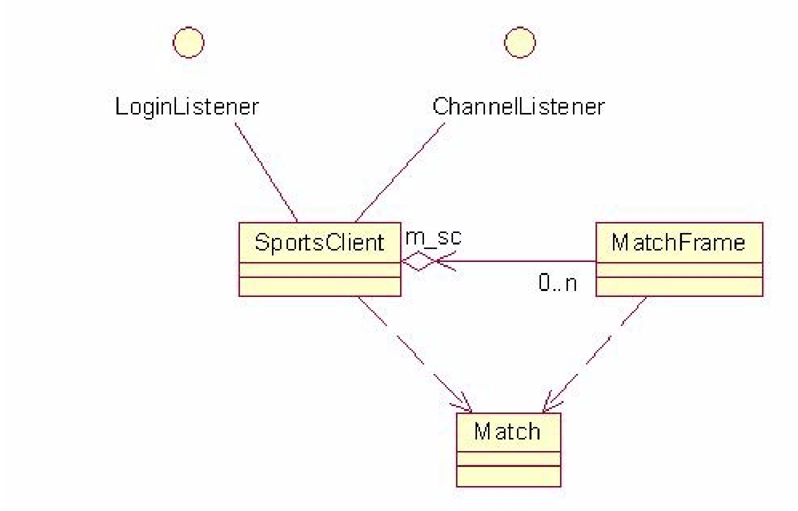


Figure 7-3 Client Class Diagram

Walkthrough: SpoortsClient also implements the login listener so it knows when it logged in to Sametime, it implements the ChannelListener so it can get an update messages from the server. It manage a hash table of MatchFrame, a MatchFrame for each subscribed match.

7.2.2 Use cases

Lets observe the main use cases and see how we solves them.

Server Logs in

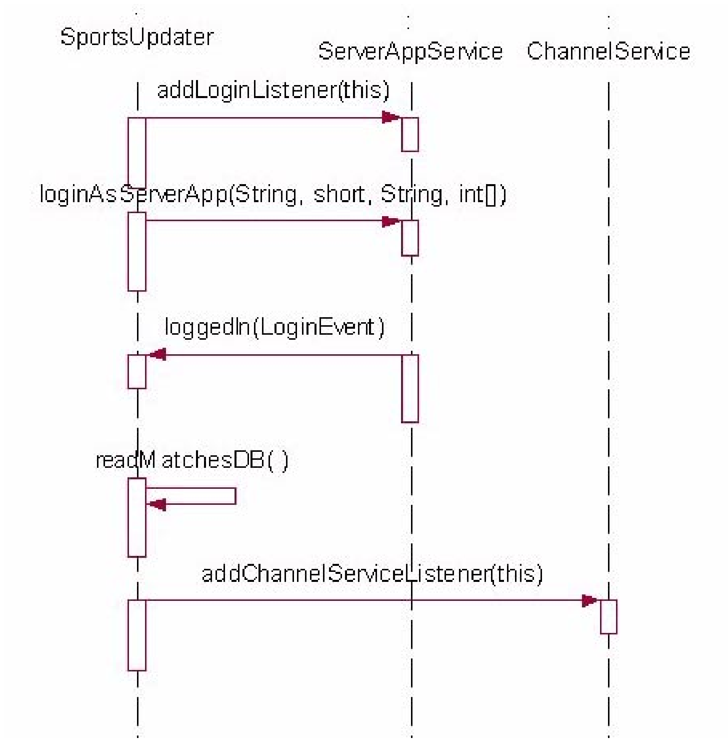


Figure 7-4 Server Logs in sequence diagram.

When the server starts, it add itself as a login listener to the ServerAppService, and uses this service to login to the community as a server application. Once it logged in it performs it specific task - in our case it reads the matches database. now it has to wait for users so it add itself to the ChannelService as a ChannelServiceListener.

Client Logs in

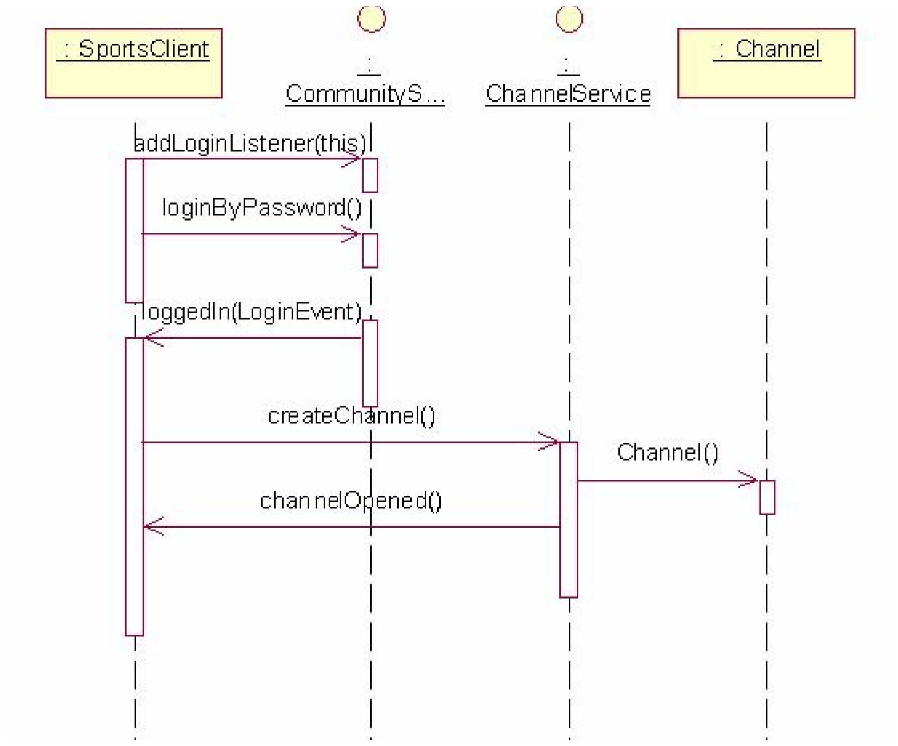


Figure 7-5 Client logs in

This is what happens when the server application starts:

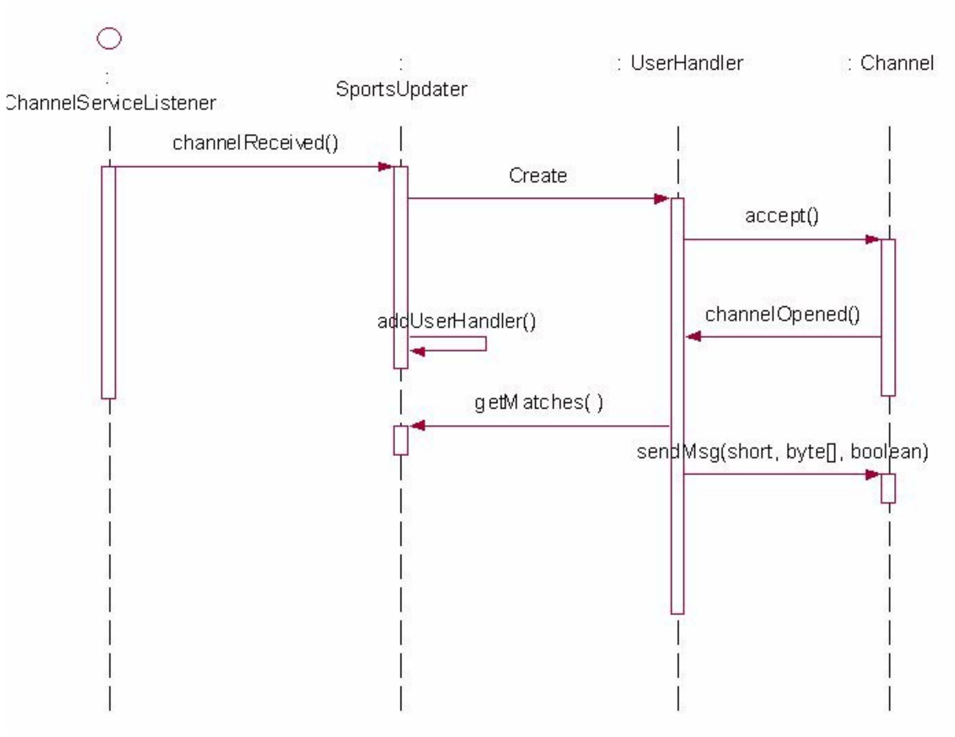


Figure 7-6 Server responds to client logs in

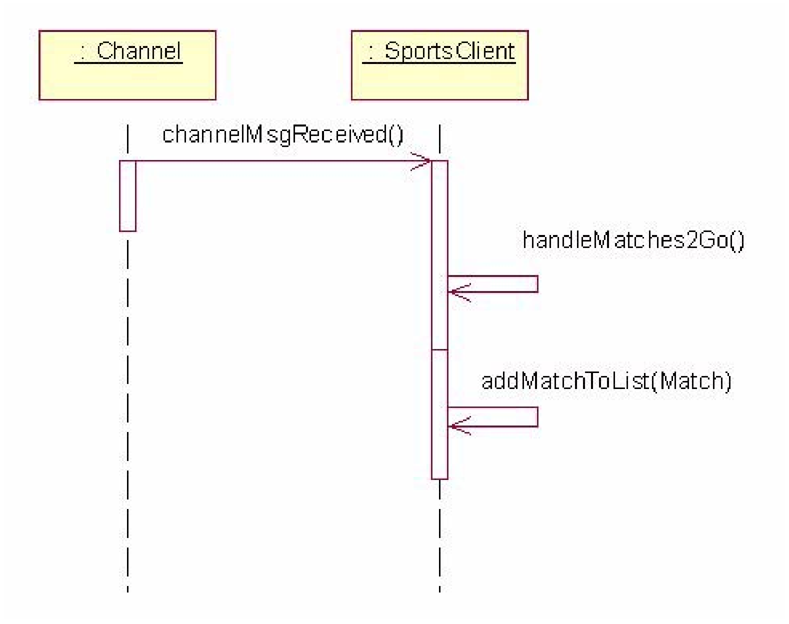


Figure 7-7 Client receive server respond

User wants to add a match

This is what happens when the user double click a list item in order to subscribe to the match in the line.

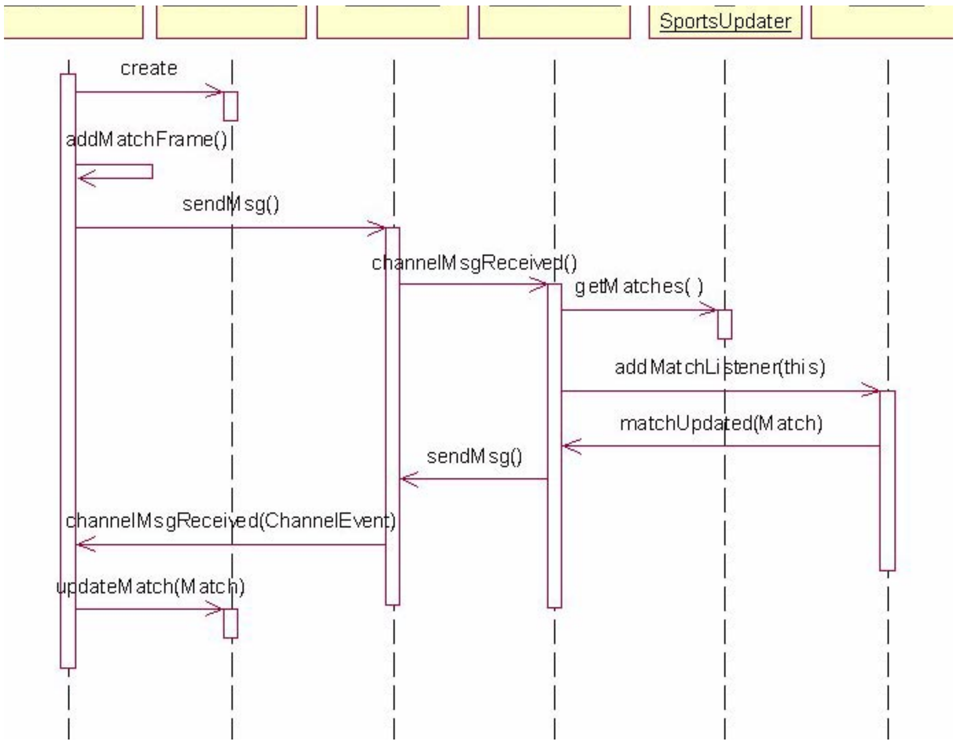


Figure 7-8 User adds a match

7.3 Implementation

By now you should have a solid idea of what we are trying to do, and how we are going to do it, so it's time for the fun part, implementation, and since this is a sample application we try to simplify the code as much as we can. During the implementation of this sample we demonstrate the following:

- Using the ServerAppService to login to Sametime as a server application, and set our Services type.
- Using the ChannelServiceListener to get an event when a user tries to open a channel to us.
- Define our own protocol.
- Streaming a class so we can transfer an object of this class between different applications.

- Sending and receiving information using a Sametime channel.

7.3.1 Configuration

Just before implementation, some small issues we must handle in order to run a Sametime server application.

Add the Sametime Server TK JAR file to your classpath.

Add stcommsrvr.jar to your classpath.

Open the Sametime server

In order to log in to the Sametime server as a server application you need to open the Sametime.ini file which is located in the Sametime folder and to perform either of the following:

1. Add your IP to the server trusted IP list, by Add/find the following line to the config section:

[Config]

VPS_TRUSTED_IPS=”trusted IP list separated by comma”

2. Configure the server to accept all IPs as trusted by Add/find the following line to the debug section:

[Debug]

VPS_BYPASS_TRUSTED_IPS=1

7.3.2 Class Match

This class represents a Macth, it has members for match details and scores, it manages a list of listeners to this match, and it knows how to stream itself.

Imports

```
import com.lotus.sametime.core.util.NdrInputStream;
import com.lotus.sametime.core.util.NdrOutputStream;
import java.util.Vector;
import java.util.Enumeration;
import java.io.IOException;
```

Class members

The class holds member for the match details, and a vector of MatchListeners:

```
// Match ID
private String m_id;
```

```
// Home team name
private String m_home;
// Visitors team name
private String m_visitors;
// Home team score
private int m_homeScore;
// Visitors team score
private int m_visitorsScore;
// Match listener list
Vector m_listeners = new Vector();
```

Constructor and get/set for class members.

In the class constructor we just initialize variables with empty values:

```
public Match()
{
   m_id = new String("");
   m_home = new String("");
   m_visitors = new String("");
   m_homeScore = 0;
   m_visitorsScore = 0;
}
```

Add a get/set methods for each member except the listeners vector (m_listeners).

Handle Match listeners

Class Match have three methods to handle MatchListeners, add, remove and update all:

```
public void addMatchListener(MatchListener listener)
{
   // add the new listener
   m_listeners.addElement(listener);

   // Send a snapshot to the listener
   listener.matchUpdated(this);
}

public void removeMatchListener(MatchListener listener)
{
   // Remove the listener
   m_listeners.removeElement(listener);
}

public void updateAllListeners()
{
   MatchListener listener;
   Enumeration enumeration = m_listeners.elements();
   while(enumeration.hasMoreElements())
   {
      listener = (MatchListener) enumeration.nextElement();
      listener.matchUpdated(this);
   }
}
```

Streaming

Since our server application sends Matches to its clients, we need away to stream Match object to a datastream. We are going to use the toolkit classes NdrOutputStream and NdrInputStream for that:

```
public void dump(NdrOutputStream  ndr) throws IOException
{
   // Write all our members into the NDR
   ndr.writeUTF(m_id);
   ndr.writeUTF(m_home);
   ndr.writeUTF(m_visitors);
   ndr.writeInt(m_homeScore);
   ndr.writeInt(m_visitorsScore);
}

public void load(NdrInputStream  ndr) throws IOException
{
   // Read all our members from the NDR
   m_id = ndr.readUTF();
   m_home = ndr.readUTF();
   m_visitors = ndr.readUTF();
   m_homeScore = ndr.readInt();
   m_visitorsScore = ndr.readInt();
}
```

7.3.3 Interface MatchListener

The MatchListener contains only one method, and it is called when the match has been updated.

```
public void matchUpdated(Match match);
```

7.3.4 Class SportsUpdater

This is the main class of the server application, it manages a Matches database, and a list of UserHandler (for each user how open a channel to it). It also responsibles to log in to the community and to gets users requests.

Imports

```
import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.core.types.*;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Random;
```

Class declaration

The SportUpdater must implements the LoginListener and the ChannelServiceListener so it could get the correct notifications from the Sametime services, it also extends thread so we can update the matches database in the backgrond.

```
public class SportsUpdater extends Thread
             implements LoginListener, ChannelServiceListener
```

Constants

Sports updater defines its service types and what message types of its protocol.

```
// Service type
public static final int SERVICE_TYPE = 0x64;

// Message types
public static final short MSG_AVAILABLE_MATCHES = 1;
public static final short MSG_SUBSCRIBE_2_MATCH = 2;
public static final short MSG_UPDATE_MATCH = 3;
public static final short MSG_UNSUBSCRIBE_FROM_MATCH = 4;
```

Class members

The class holds member for the match details, and a vector of MatchListeners:

```
// The Sametime toolkit session
private STSession m_session;
// The server application service
private ServerAppService m_saService;
// The matches list
Vector m_users = new Vector();
// The user handles list
Vector m_matches = new Vector();
//  Random to make some changes in the matches
Random m_rnd = new Random();
```

Login to Sametime

The first we want to do as a server application is to login to the community, so, in the initialize method we loads all component and logs in, we take the server name from the command line:

```
public static void main(String[] args)
{
   if ( args.length != 1 )
```

```
        {
            System.out.println("Usage: SportUpdaterSA serverName");
            System.exit(0);
        }

            new SportsUpdater().initialize(args[0]);
    }

    private void initialize(String serverName)
    {
        // init the sametime session and objects
        try
        {
            // First, we create a new session, that belongs uniquely to us.
            m_session = new STSession("" + this );

            // Load components, we only need STBase for login and channel
            m_saService = new STBase(m_session);

            // start the session
            m_session.start();
        }
        catch (DuplicateObjectException e)
        {
            System.out.println("STSession or Components created twice.");
            e.printStackTrace();
           System.exit(1);
        }

        // Now we can login
        login(serverName);
    }

    private void login(String serverName)
    {
        // Add ourselves as login listener to the community
        m_saService.addLoginListener( this);

        // Login to sametime as a server application, pass our ServiceType
        int[] supportedServices = { SERVICE_TYPE };

        short loginType = STUserInstance.LT_SERVER_APP;
        m_saService.loginAsServerApp( serverName, loginType,
                                 "Sports Updater", supportedServices);
    }
```

Implement LoginListener

We get the server respond to login request by implementing the LoginListener, if the server rejects our login request we would get a loggedOut event, and if the server accept we would get a loggedIn event. So, the loggedIn event is the right place to start perform our task, i.e. read the matches database, start the updating thread, and wait for user requests.

```
public void loggedIn(LoginEvent event)
{
   // Read the matches database
   readMatchesDB();

   // Start the updating thread
   start();

   // Get a ref to the channel service
   ChannelService channelService =
         (ChannelService)m_session.getCompApi(ChannelService.COMP_NAME);

   channelService.addChannelServiceListener(this);
}

public void loggedOut(LoginEvent event)
{
   System.out.println("Loged out, reason = " + event.getReason() );
}
```

Matches database

In a real world application we would probably create the Match database by reading it from the web (using webservices for example), however in order to simplify this sample we just create the matches hard coded, and update the scores using the random class.

So we need methods to create the database, to update it from another thread (refresh and refreshMatch), we also have a getMatches method which returns a copy of the database.

```
private void readMatchesDB()
{
   Match m = new Match();
   m.setID("01");
   m.setHomeTeamName("Philadelphia");
   m.setVisitorsTeamName("Miami");
   m_matches.addElement(m);

   m = new Match();
```

```
        m.setID("02");
        m.setHomeTeamName("New Jersay");
        m.setVisitorsTeamName("Indiana");
        m_matches.addElement(m);

        m = new Match();
        m.setID("03");
        m.setHomeTeamName("Houston");
        m.setVisitorsTeamName("San Antonio");
        m_matches.addElement(m);

    }

    public Vector getMatches()
    {
        // Return a clone to the real matches
        return (Vector)m_matches.clone();
    }

    public void run()
    {
        while ( true )
        {
            // Check for changes in the database
            refresh();

            // go to sleep for a while
            try
            {
                sleep(1000);
            }
            catch( InterruptedException exc)
            {
            }
        }
    }

    private void refresh()
    {
        Match m;
        Enumeration enumeration = m_matches.elements();
        while(enumeration.hasMoreElements())
        {
             m = (Match) enumeration.nextElement();
            refreshMatch(m);
        }
    }
```

```
private void refreshMatch(Match m)
{
   // Do we want to change this match at all
   if ( (m_rnd.nextInt() % 3) != 0 )
      return;

   // Randomally add between zero to three points to home team
   if ( (m_rnd.nextInt() % 3 == 0 ) )
      m.setHomeTeamScore( m.getHomeTeamScore() + 2 );
   else if ( (m_rnd.nextInt() % 5 == 0 ) )
      m.setHomeTeamScore( m.getHomeTeamScore() + 1 );
   else if ( (m_rnd.nextInt() % 5 == 0 ) )
      m.setHomeTeamScore( m.getHomeTeamScore() + 3 );

   // Randomally add between zero to three points to visitors
   if ( (m_rnd.nextInt() % 3 == 0 ) )
      m.setVisitorsTeamScore( m.getVisitorsTeamScore() + 2 );
   else if ( (m_rnd.nextInt() % 5 == 0 ) )
      m.setVisitorsTeamScore( m.getVisitorsTeamScore() + 1 );
   else if ( (m_rnd.nextInt() % 5 == 0 ) )
      m.setVisitorsTeamScore( m.getVisitorsTeamScore() + 3 );

   m.updateAllListeners();
}
```

Handling of user requests

When the we logs in, we sets our service type, so the Sametime server knows to address channel with this service type to us. In the loggedIn event we add ourself as ChannelServiceListener to the ChannelService, we have to implement this interface which contains only one method: ChannelRecieved. This event is called everytime a user tries to open a channel to us, so in the implementation we just creates a UserHandler for this user and let it to the work. We also implement a remove user handler method so the user handler can remove itself when its user close the channel.

```
public void channelReceived(ChannelEvent event)
{
   // Create a user handler for this user, and it it to our list
   UserHandler user = new UserHandler(this, event);
   addUserHandler(user);
}

public void addUserHandler(UserHandler user)
{
   // add the new handler
   m_users.addElement(user);

public void removeUserHandler(UserHandler user)
```

```
{
   // remove the user listener
   m_users.removeElement(user);
}
```

7.3.5 Class UserHandler

This class handles one user request, when the user opens a channel to the SportsUpdater the SportUpdater creates a UserHandler for this user and pass it the Channel details. The UserHandler add itself as a listener to the channel and accept it. When the channelOpened event arrives the handler sends the user a snapshot of all matches and keep listen to the channel. Whenever a user sends a subscribe/unsubscribe match message, the handler adds/removes itself as listener to this match. When the match changed it gets a notification and sends a match update message to the user.

Class declaration

```
public class UserHandler implements ChannelListener, MatchListener
```

Class members

The class holds reference to the SportUpdater, a channel to the user and a vector of Match IDs.

```
// Reference to the updaterr
private SportsUpdater m_updater;
// The channel to the user
private Channel m_cnl;
// vector of subscribed match IDs
   private Vector m_matches = new Vector();
```

Initilization

The constructor has two parameters, the SportsUpdater which created us, and ChannelEvent object which contains the Channel which our user tried to open to us, so we just keep the information and accept the channel, the real work is done in the implementation of the ChannelListener.

```
public UserHandler(SportsUpdater updater, ChannelEvent event)
{
   // Save ref to parent
   m_updater = updater;

    // Get the channel
   m_cnl = event.getChannel();
```

```
    // Add ourselve as listener and accept the channel
    m_cnl.addChannelListener(this);
    m_cnl.accept(EncLevel.ENC_LEVEL_DONT_CARE, null);
}
```

Implement ChannelListener

ChannelListener contains four methods: channelOpened, channelOpenFailed, channelClosed and channelMsgReceived. In the channelOpened method we send a snapshot of the available matches to the user. If the channel failed to open (ChannelOpenFailed we just print a message. In the channelClosed event we have to do a clean up. In the channelMsgReceived we have to determine what type of message is it and to register/unregister in correspond.

```
public void channelOpened(ChannelEvent event)
{
    // Get a copy of all the matches from the upgrader
    Vector matches = m_updater.getMatches();

    // Dump the matches
    NdrOutputStream ndr = new NdrOutputStream();
    try
    {
        // Dump number of matches
        int i = matches.size();
        ndr.writeInt(i);

        // Dump all matches
        Match m;
        Enumeration enumeration = matches.elements();
        while(enumeration.hasMoreElements())
        {
            m = (Match) enumeration.nextElement();
            m.dump(ndr);
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
        return;
    }

    // Send the message
    m_cnl.sendMsg(SportsUpdater.MSG_AVAILABLE_MATCHES,
                                     ndr.toByteArray(), false);
}

public void channelOpenFailed(ChannelEvent event)
```

```
{
   System.out.println("channelOpenFailed");
   m_updater.removeUserHandler(this);
}

public void channelClosed(ChannelEvent event)
{
   // Unsubscribe from all matches
   Match m;
   Vector temp = m_matches;
   Enumeration enumeration = temp.elements();
   while(enumeration.hasMoreElements())
   {
      m = (Match) enumeration.nextElement();
      m.removeMatchListener(this);
   }
   // Remove ourselves from the updater records
   m_updater.removeUserHandler(this);
}

public void channelMsgReceived(ChannelEvent event)
{
   int msgType = event.getMessageType();
   switch (msgType)
   {
      case SportsUpdater.MSG_SUBSCRIBE_2_MATCH:
         handleSubscribe(event, true);
         break;
      case SportsUpdater.MSG_UNSUBSCRIBE_FROM_MATCH:
         handleSubscribe(event, false);
         break;
      case SportsUpdater.MSG_AVAILABLE_MATCHES:
      case SportsUpdater.MSG_UPDATE_MATCH:
      default:
         System.out.println("Error in UserHandler::channelMsgReceived");
         break;
   };
}

private void handleSubscribe(ChannelEvent event, boolean subscribe)
{
   String matchID;
   try
   {
      // This message only contains the match ID
      NdrInputStream ndr = new NdrInputStream(event.getData());
      matchID = ndr.readUTF();
   }
   catch (IOException e)
```

```
    {
       e.printStackTrace();
       event.getChannel().close(STError.ST_INVALID_DATA, null);
       return;
    }

          // Subscribe/unsubscribe to this match
          subscribeToMatch(matchID, subscribe);


private void subscribeToMatch(String matchID, boolean subscribe)
{
    // Get a copy of all the matches from the updater
    Vector matches = m_updater.getMatches();
    Enumeration enumeration = matches.elements();
       // Walk through all matches
    Match m;
    while(enumeration.hasMoreElements())
    {
       // Get next match
        m = (Match) enumeration.nextElement();

        // Our match?
        if ( m.getID().equals( matchID ) )
        {
           if ( subscribe )
           {
              // subscribe to the match and add it to our records
              m.addMatchListener(this);
              m_matches.addElement(m);
           }
           else
           {
              // Unsubscribe and remove the match from our records
              m.removeMatchListener(this);
              m_matches.removeElement(this);
           }
        }
    }
}
```

Implement MatchListener

This interface contains only one method, matchUpdated, and in the implementation of this function we are going to send the updated match to the client.

```
public void matchUpdated(Match match)
{
   // Dump the match to a NDR
   NdrOutputStream ndr = new NdrOutputStream();
   try
   {
      match.dump(ndr);
   }
   catch(IOException e)
   {
      e.printStackTrace();
      System.out.println("Error dumping Match");
      return;
   }

   // Send the message
   if ( m_cnl.isOpen() )
   {
      m_cnl.sendMsg(SportsUpdater.MSG_UPDATE_MATCH,
                                ndr.toByteArray(), false);
   }
}
```

7.3.6 Class SportsClient

This is the main class of the client application, it responsible to show a window with a list of available matches and to open a match window when the user double click on one of the matches in the list.

Imports

```
import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.core.util.NdrOutputStream;
import com.lotus.sametime.core.util.NdrInputStream;
import com.lotus.sametime.core.constants.*;
import com.lotus.sametime.core.types.*;

import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.StringTokenizer;
```

Class declaration

We need a main window so we make him extends Frame. We also needs to know when we logged in so SportsClient should implements the LoginListener and it also implements the ChannelListener so we can get notifications from the SportsUpdater. The last interface implemented is the ActionListener so we can opens a window when the user double click on a list item.

```
public class SportsClient extends Frame
      implements LoginListener, ChannelListener, ActionListener
```

Class members

From the Sametime toolkit we need the STSession and a channel so we can communicate with the SportsUpdater server application. Since every match has a unique ID we keep a hash table of MatchFrame storing by the Match ID key. We also have a List to display the available matches.

```
// Sametime session object.
private STSession m_session;
// channel to the user
Channel m_cnl;
// matches list
private List m_lst;
// The MatchFrame dictionary (for each subscribed match we have a frame)
private Hashtable m_frames = new Hashtable();
```

Login to Sametime

The SportsClient gets the server name, user name and password from the command line and uses them to login to Sametime when the application starts.

```
public static void main(String[] args)
{
   if (args.length != 3)
   {
      System.err.println("Usage: Client [name] [password] [serverName]");
      System.exit(0);
   }

   new SportsClient().run(args[0], args[1], args[2]);
}

public SportsClient()
{
   super("Sports Client");
}

private void run(String name, String password, String server)
{
   // Create and load the session of components.
   try
   {
      m_session = new STSession("SportsUpdaterClient");
      m_session.loadAllComponents();
      m_session.start();
   }
   catch(DuplicateObjectException e)
   {
      e.printStackTrace();
      System.exit(1);
   }

   // Login to the community
   login(name, password, server);

}


void login(String name, String password, String hostName)
{
   CommunityService commService
         = CommunityService)m_session.getCompApi(CommunityService.COMP_NAME);

   commService.addLoginListener(this);
   commService.loginByPassword(hostName, name, password);
}
```

Implement LoginListener

Once we logged in we want to create a channel to the server application.

```
public void loggedIn(LoginEvent event)
{
     // Get a reference to the channel services.
    ChannelService channelService =
                (ChannelService)m_session.getCompApi(ChannelService.COMP_NAME);

    // Create a channel to the sports updater SA
    m_cnl = channelService.createChannel(SportsUpdater.SERVICE_TYPE, 0, 0,
                                          EncLevel.ENC_LEVEL_ALL, null, null);

    // Listen to this channel
    m_cnl.addChannelListener(this);

    // Open, the channel
    m_cnl.open();
}

public void loggedOut(LoginEvent event)
```

```
{
   System.out.println("Logged out, reason = " + event.getReason() );
   exit();
}
```

Implement ChannelListener

Getting the channelOpened event means that the server application accepted our channel and is about to send us a snapshot message, this is a good place to create the UI. If we fail to create the channel or the channel is closed we just print a message to the console and exit. In the channelMsgReceived we just check the message type and process the message.

```
public void channelOpened(ChannelEvent event)
{
   // The channel is opened, create the UI
   createUI();
}

public void channelOpenFailed(ChannelEvent event)
{
   System.out.println("SportClent::channelOpenFailed");
   System.out.println("Server Application is probably not running");

   exit();
}

public void channelClosed(ChannelEvent event)
{
   System.out.println("SportClent::channelClosed");
   System.out.println("Reason = " + event.getReason() );

   exit();
}


public void channelMsgReceived(ChannelEvent event)
{
   // What kind of message did we get?
   int msgType = event.getMessageType();
   switch ( msgType )
   {
      case SportsUpdater.MSG_AVAILABLE_MATCHES:
         handleAvailableMatches(event);
         break;
      case SportsUpdater.MSG_UPDATE_MATCH:
         handleUpdateMatch(event);
         break;
      case SportsUpdater.MSG_SUBSCRIBE_2_MATCH:
```

```
      case SportsUpdater.MSG_UNSUBSCRIBE_FROM_MATCH:
      default:
         System.out.println("Error in SportsClient::channelMsgReceived");
         break;
   };
}
```

Handling messages

In the client we have to handle the available matches message and the update match message, in the first we just add all the messages to the list box, and in the second, we find the MatchFrame and pass it the message.

The protocol allows us to subscribe/unsubscribe to a match, since the structures of this messages are similar we implement them in the same method.

```
private void handleAvailableMatches(ChannelEvent event)
{
   try
   {
      // Get the data
      NdrInputStream ndr = new NdrInputStream(event.getData());

      // How many matches do we have?
      int size = ndr.readInt();

      // Read all matches
      for (int i = 0; i < size; i++)
      {
         Match m = new Match();
         m.load(ndr);

         // Add each matches to our list
         addMatchToList(m);
      }
   }
   catch (IOException e)
   {
      e.printStackTrace();
      event.getChannel().close(STError.ST_INVALID_DATA, null);
      return;
   }
}

private void handleUpdateMatch(ChannelEvent event)
{
   // Get the data
   NdrInputStream ndr = new NdrInputStream(event.getData());
```

```
        // Read the match details
        Match m = new Match();
        try
        {
            m.load(ndr);
        }
        catch(IOException e)
        {
            e.printStackTrace();
            System.out.println("Error loading Match");
            return;
        }

        // Find the match frame
        MatchFrame f = (MatchFrame)m_frames.get( m.getID() );

        // Update frame
        f.updateMatch( m );
    }

    private void subscribeToMatch(String id, boolean subscribe)
    {
        // Write the mathc ID on the NDR
        NdrOutputStream ndr = new NdrOutputStream();
        try
        {
            ndr.writeUTF(id);
        }
        catch(IOException e)
        {
            e.printStackTrace();
            System.out.println("Error in subscribeToMatch, could not dump matches");
            return;
        }

        // Are we subscribing or unsubscribing
        short cmd = subscribe ? SportsUpdater.MSG_SUBSCRIBE_2_MATCH :
                    SportsUpdater.MSG_UNSUBSCRIBE_FROM_MATCH;

        // Send the message
        m_cnl.sendMsg(cmd, ndr.toByteArray(), false);
    }
```

UI

In createUI we just create all the UI objects and register listener when needed. And the actionPerformed, we find the match the user just double click and add a match frame for it.

In addMatchToList we just add a line with the match details to the list.

```
private void  createUI()
{
   m_lst = new List();
   m_lst.addActionListener(this);
   add(m_lst);
   setSize(200, 300);

   Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
   setLocation(
   ( screen.width - getSize().width ) / 2,
   ( screen.height - getSize().height ) / 2 );
   setVisible(true);


   addWindowListener(
                        new WindowAdapter()
                        {
                           public void windowClosing(WindowEvent event)
                           {
                              exit();
                           }
                        }
                  );
}

   /**
    * User double click a list item
    */
   public void actionPerformed(ActionEvent event)
    {
      // Get the line
      String s = event.getActionCommand();
      if ( s.length() == 0 )
         return;

      // Get the first token from the line (match ID)
      StringTokenizer st = new StringTokenizer(s);
      s = st.nextToken();

      // Ok, add a new MatchFrame to our dictionary
      addMatchFrame(s);
    }
}

private void addMatchToList(Match m)
{
   // Create a string for this match
   String s ;
   s = m.getID() +  " " + m.getHomeTeamName() +
                " Vs. " + m.getVisitorsTeamName();

   // Add it to our list
   m_lst.add(s);
}
```

Handling Match Frame

We need to add and remove MatchFrame to our hash table, and to subscribe/unsubscribe from the match when doing do.

When also have an exit function to do a clean up while the application is being terminated.

```
private synchronized void addMatchFrame(String matchID)
{
   // Create a new MatchFrame and add it to our hashtable
   Hashtable temp = (Hashtable) m_frames.clone();
   temp.put( matchID, new MatchFrame(this, matchID) );
   m_frames = temp;

   // Subscribe to this match
   subscribeToMatch(matchID, true );
}

public synchronized void removeMatchFrame(String matchID)
{
   // Remove it from out hashtable
   Hashtable temp = (Hashtable) m_frames.clone();
   temp.remove(matchID);
   m_frames = temp;

   // Unsubscribe to this match
   subscribeToMatch(matchID, false);
}

private void exit()
{
   // Unload the Sametime components and session
   m_session.stop();
   m_session.unloadSession();

   // Close all frames
   MatchFrame f;
   Hashtable temp = (Hashtable)m_frames.clone();
   Enumeration enumeration = temp.elements();
```

```
    while(enumeration.hasMoreElements())
    {
        f = (MatchFrame) enumeration.nextElement();
        f.dispose();
    }

    // Exit
    dispose();
    System.exit(0);
}
```

7.3.7 Class MatchFrame

This class manage the UI for one match.

Imports

```
import java.awt.*;
import java.awt.event.*;
```

Class declaration

MatchFrame extends Frame.

```
public class MatchFrame extends Frame
```

Class members

We just keep a reference to the SportsClient which created us so we can remove ourself while closing. We keep our match ID, and some UI elements.

```
// The owner sports client
SportsClient m_sc;
// The Match ID
String m_id;
// Home scores
// Label m_lblHomeScore;
// Visitors score
// Label m_lblVisitorsScore;
// Initial left position
static int m_left = 10;
// Initial top position
static int m_top = 10;
```

Implementation

The class contains constructor, create UI method when we create the UI, and an updateMatch method to update the UI.

```
public MatchFrame(SportsClient client, String id)
```

```
{
   m_sc = client;
   m_id = id;

   createUI();
}

private void createUI()
{
   setLayout(new BorderLayout());

   m_lblHomeScore = new Label("Home:");
   add(m_lblHomeScore, BorderLayout.NORTH );

   m_lblVisitorsScore = new Label("Visitors:");
   add(m_lblVisitorsScore, BorderLayout.SOUTH );

   setSize(100, 100);
   setLocation( m_left, m_top );
   m_left += 20;
   m_top += 20;
   setVisible(true);

   addWindowListener(
               new WindowAdapter()
                         {
                            public void windowClosing(WindowEvent event)
                            {
                               m_sc.removeMatchFrame(m_id);
                               dispose();
                            }
                         }
                  );
}


public void updateMatch(Match match)
{
   // Do we have something to update?
   if ( match.getHomeTeamName().length() == 0 )
      return;

   // Title should be set only once
   if ( getTitle().length() == 0 )
   {
      setTitle( match.getHomeTeamName() + "Vs. " +
               match.getVisitorsTeamName() );
   }
```

```
    // Update score
    String s = match.getHomeTeamName() + ": " + match.getHomeTeamScore();
    m_lblHomeScore.setText(s);

    s = match.getVisitorsTeamName() + ": " + match.getVisitorsTeamScore();
    m_lblVisitorsScore.setText(s);
}
```

7.4 Running

The next step is to run and test our applications, so first thing you do is to press the build button of your IDE.

Since we have two applications in our project we need to run one of them from the command line or from a batch file, let write two batch files, one to run the server and one to run the client.

Note: This description assumes you have Visual J++ installed. We will update it with instructions for a common JDK.

- RunSvr.bat

```
cd <your output folder path>
jview /cp <server toolkit jar path> SportsUpdater <server name>
```

- RunClient.bat

```
cd <your output folder path>
jview /cp <server toolkit jar path> SportsClient <user name> <password>
<server name>
```

All you have to do now is to start the server application by activating the RunSvr.Bat and then start the client by activating the Runclient.bat, enjoy.

7.5 How to make this sample a real world application.

The application we just created is far away from being a real world application, if you have an idea of a service you want to supply using the sample mechanism here are a few suggestions of how to make this sample a real world application.

- Redefine the protocol between the server and the client so it fits your need:
  - The server sends a matches available message whenever a client opens a channel to it, this is done just to save some code. In a real world the client should send a request of what it needs to the server first.

   - Add messages types so you can select which kind of information the client is looking for.
   - Define much more effective protocol, for example in the sample we send a stream of a match both for the "matches available" message and the "match updated" message, this makes us less effective. why sending the match score in a "matches available" message, and why sending the team names in a "match updated" message.
- Do something about the embarrassing UI (no need to explain this, right?).
- Other creative ideas.
   - You can supply an HTML base interface for matches registration, and then use the server toolkit to send an IM to the registered user.
   - You can add teams as users to the directory, and when the match starts use the LoghtLoginService to change the status of the user to active. Users can add their favourite team to their buddy list. And when the match starts the team becomes online.

Part 1

Appendixes

A

Additional material

No additional Web material is included with this redpiece. Most of the code we discuss is included with the beta version of the Sametime Community Server Toolkit which is available for download from

```
http://www.lotus.com/sametimedevelopers
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 111.

- *Working with the Sametime Client Toolkits, SG24-6666*
- *Lotus Sametime 2.0 Deployment Guide, SG24-6206*
- *B2B Collaborative Commerce with Sametime, QuickPlace and WebSphere Commerce Suite, SG24-6218*
- *Applying the Patterns for e-business to Domino and WebSphere Scenarios, SG24-6255*
- *Lotus Mobile and Wireless Solutions, SG24-6525*
- *Customizing QuickPlace, SG24-6000*
- *Domino and WebSphere Together - Second Edition, SG24-5955*
- *COM Together with Domino, SG24-5670*
- *Lotus Domino Release 5.0: A Developer's Handbook, SG24-5331*
- *iNotes Web Access Deployment and Administration, SG24-6518*
- *Inside the Lotus Discovery Server, SG24-6252*
- *XML Powered by Domino How to use XML with Lotus Domino, SG24-6207*
- *Using VisualAge for Java to Develop Domino Applications, SG24-5424*
- *Connecting Domino to the Enterprise Using Java, SG24-5425*
- *Performance Considerations for Domino Applications, SG24-5602*
- *Using Domino Workflow, SG24-5963*

Other resources

These publications are also relevant as further information sources:

- *????full title???????, xxxx-xxxx*

- *????full title???????, xxxx-xxxx*
- *????full title???????, xxxx-xxxx*

Referenced Web sites

These Web sites are also relevant as further information sources:

- The developers corner of the Sametime Web site. Here you can download toolkits, beta version, browse technical documents and participate in the Sametime developers discussion forum

  `http://www.lotus.com/sametimedevelopers`
- Lotus Developer Network is Lotus' primary destination for the latest developer information and resources. Contains articles about new and current technologies along with relevant tips and techniques to help you build dynamic collaborative e-business applications.

  `http://www.lotus.com/developer/`
- Notes.net from Iris, the developers of Notes and Domino, is a technical Web site with discussion forums, documentation and the Webzine "Iris Today", with many good articles about technical details of Domino.

  `http://notes.net/`
- The IBM developerWorks Web site is designed for software developers, and features links to a host of developer tools, resources, and programs.

  `http://ibm.com/developer/`
- Lotus Support's Web site - Search using keywords or browse the Lotus Knowledge Base and locate helpful and informative tech notes and technical papers for the entire Lotus Product family. This source of information contains the latest technical information updated hourly.

  `http://support.lotus.com/`
- Entry point to information about the IBM WebSphere software platform for e-business

  `http://ibm.com/websphere/`

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5” spine. Now select the Spine width for the book and hide the others: **Special>Conditional Text>Show/Hide>SpineSize(-->Hide:)>Set**

IBM Redbooks

Working with the Sametime Community Server Toolkit

(1.5” spine)
1.5”<-> 1.998”
789 <->1051 pages

IBM Redbooks

Working with the Sametime Community

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

IBM Redbooks

Working with the Sametime Community Server Toolkit

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

IBM Redbooks

Working with the Sametime Community Server Toolkit

(0.2”spine)
0.17”<->0.473”
90<->249 pages

(0.1”spine)
0.1”<->0.169”
53<->89 pages

To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5” spine. Now select the Spine width for the book and hide the others: **Special>Conditional Text>Show/Hide>SpineSize(-->Hide:)>Set**

IBM

Redbooks

Working with the Sametime Community Server Toolkit

(2.5” spine)
2.5”<->nnn.n”
1315<-> nnnn pages

IBM

Redbooks

Working with the Sametime Community Server Toolkit

(2.0” spine)
2.0” <-> 2.498”
1052 <-> 1314 pages

Working with the Sametime Community Server Toolkit

Enhance and extend Sametime services

Create your own Sametime service

Support new clients and more

The ability to create Sametime server side applications dramatically increase the potential for great benefits from using real time collaborations in all parts of businesses and organizations.
This WhitepaperRedpaperredpiece contains some of the content that will be included in the upcoming Redbook about the Sametime Community Server Toolkit. It is being released in a very early stage to provide additional information for the users of the beta version of the Sametime Community Server Toolkit.
We introduce the Sametime server architecture which you need to understnad to get the most out of server side programming.
We discuss two of the samples included with the toolkit called HackersCatcher and PlacesLogger.
Finally we take you through how to create a new Sametime service called SportsUpdater. This is also included as a sample in the beta version of the toolkit.

SG24-6667-00 ISBN 0738423912